图灵程序设计丛书

PACKT PUBLISHING

[意] 马尔科 · 邦扎尼尼 著　陈小莉 陶俊杰 译

Python 社会媒体挖掘

Mastering Social Media Mining with Python

获取、存储、分析和可视化社交数据的一站式解决方案参考指南



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

马尔科 · 邦扎尼尼
（Marco Bonzanini）
数据科学咨询师，拥有伦敦玛丽王后大学信息检索专业博士学位，是PyData伦敦meetup及系列会议的合作组织者，在很多国际会议上做过演讲，并且在PacktPub上教授“Python数据分析”和“实用Python数据科学技术”两门课程。他在个人博客上分享了很多技术主题，主要关于Python、文本分析和数据科学。

数字版权声明



图灵程序设计丛书

Python 社会媒体挖掘

Mastering Social Media Mining with Python

[意] 马尔科 · 邦扎尼尼 著
陈小莉 陶俊杰 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Python社会媒体挖掘 / (意) 马尔科·邦扎尼尼(Marco Bonzanini) 著 ; 陈小莉, 陶俊杰译. -- 北京 : 人民邮电出版社, 2018.10
（图灵程序设计丛书）
ISBN 978-7-115-49401-6

Ⅰ. ①P… Ⅱ. ①马… ②陈… ③陶… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第216714号

内 容 提 要

本书共分为 9 章，从社会媒体 API、数据挖掘技巧和 Python 的数据科学工具这 3 个主题进行阐释。主要内容包括：如何用 Python 通过公共 API 与社会媒体平台交互，如何以方便的格式为数据分析存储社会媒体数据，如何使用 Python 数据科学工具分割社会媒体数据，如何用文本分析方法理解社会媒体数据，如何用先进的统计和分析手段从海量数据中挖掘出有用信息，以及如何用 Web 技术来可视化数据。

本书适合社会媒体挖掘相关从业人员阅读。

◆ 著　[意] 马尔科·邦扎尼尼
译　陈小莉　陶俊杰
责任编辑　杨　琳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：15
字数：373千字　2018年 10 月第 1 版
印数：1 - 3 000册　2018年 10 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-4175号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

前　　言

近年来，社会媒体的流行程度猛增，越来越多的用户正在通过不同的平台分享信息。公司用社会媒体平台进行品牌营销，专业人士在线维护其公众形象并用社会媒体拓展人脉，而普通用户则在社会媒体上讨论任意主题。更多的用户，就意味着有更多的数据等待挖掘。

本书的读者可能是开发者、工程师、分析师、研究人员，或者是希望将数据挖掘技术应用于社会媒体数据的学生。从数据丰富度的角度来看，数据挖掘实践者（或者是未来的实践者）并不缺少机会和挑战。

本书将介绍充分利用这些数据所需的基本工具。书中首先介绍用于 Python 数据分析的主要工具，提供一些信息供你上手自然语言处理、机器学习、社会网络分析和数据可视化等应用。接着手把手地指导你如何从 Twitter、Facebook、Google+、Stack Overflow、Blogger、YouTube 等最流行的社会媒体平台获取数据，以及如何进行不同类型的分析，以便从原始数据中提炼有价值的洞见。

本书主要涉及三个主题，如下所示。

- **社会媒体 API**：每个平台都提供了不同的数据获取方式。理解如何与它们交互可以回答以下问题：**如何获取数据？可以获取哪种类型的数据？**这些问题非常重要，因为如果没有获取到数据，就无法进行数据分析。每一章会重点介绍一个不同的社会媒体平台，并详细介绍如何与相关的 API 进行交互。
- **数据挖掘技巧**：仅仅是从 API 获取数据并没有太大价值。下一步需要回答：**可以用这些数据做什么？**每一章将介绍对相应数据进行不同类型的分析所需掌握的概念，以及为什么这些分析能够带来价值。本书仅对相关理论进行浅显的介绍，深入介绍请参见相应的学术著作。本书的目的是为读者提供一些可以迅速上手的示例。
- **Python 的数据科学工具**：搞清楚可以用数据做什么后，最后一个问题是：**如何做？** Python 是数据科学的主流语言之一，其语法和语义简单易懂，并且拥有丰富的科学计算生态系统，不仅对初学者来说学习曲线非常平滑，而且为专家提供了专业的工具。本书介绍了用于科学计算的主要 Python 库，如 NumPy、pandas、NetworkX、scikit-learn、NLTK 等。实例将提供精简的代码，以帮助你完成各种有趣的社会媒体数据分析。

如果对以上三个方面感兴趣，那么本书就很适合你。

本书内容

第 1 章，社会媒体、社交数据和 Python。这一章将介绍用 Python 进行社会媒体数据挖掘的基本概念。通过简要介绍机器学习、自然语言处理、社会网络分析和数据可视化，这一章将介绍 Python 主要的数据科学工具，以及 Python 开发环境的安装方法。

第 2 章，Twitter 数据挖掘——标签、话题和时间序列。这一章介绍如何用 Twitter 数据进行数据挖掘。首先设置一个与 Twitter API 交互的 Twitter 应用，然后介绍如何用流 API 获取数据，以及如何对话题标签和推文进行频率分析。此外还将介绍一些时间序列分析方法，以理解推文随时间的分布情况。

第 3 章，Twitter 用户、粉丝和社区。这一章将继续介绍 Twitter 挖掘，重点关注用户及用户间的互动。我们将演示如何挖掘用户间的联系与对话，还将介绍一些有趣的应用，其中包括用户聚类（分组），以及用户影响力与参与度的度量方法。

第 4 章，Facebook 帖子、页面和用户互动。这一章将重点介绍 Facebook 和 Facebook Graph API。首先介绍 Facebook Graph API 的交互方式，包括隐私和安全问题，然后用示例演示如何挖掘用户信息页和 Facebook 页面的帖子。我们将用时间序列和用户参与度的概念来分析用户的互动行为，其中包括评论、喜欢和 Reactions。

第 5 章，Google+话题分析。这一章将介绍 Google 的社交网络。首先介绍如何接入 Google 的中心化平台，然后用示例演示如何在 Google+中搜索内容和用户。此外，还会演示如何将 Google API 的数据嵌入一个由 Python 的微框架 Flask 建立的 Web 应用。

第 6 章，Stack Exchange 提问和回答。这一章将介绍问答类主题网站，并将 Stack Exchange 网络作为主要示例。你将学会如何搜索这个网络中不同站点的用户和内容，尤其是 Stack Overflow。借助它们的存档数据和在线处理功能，这一章还将介绍监督机器学习方法在文本分类方面的应用，并演示如何在实时应用中嵌入机器学习模型。

第 7 章，博客、RSS、维基百科和自然语言处理。这一章将介绍文本分析方法。Web 中充满了文本挖掘的机会，这一章将演示如何与 WordPress.com API、Blogger API、RSS 订阅和维基百科 API 等数据源进行交互。之后利用文本数据，对之前简单提及的自然语言处理的基本概念进行正式阐述与扩展。然后介绍信息抽取过程，并用示例演示如何从自由文本中抽取实体。

第 8 章，挖掘所有数据。在最常用的社交网络之外还有许多数据挖掘机会。这一章用示例演示了如何挖掘 YouTube、GitHub 和 Yelp 数据，并探讨了当平台没有提供 API 时，如何构建自己的 API 客户端。

第 9 章，关联数据和语义网。这一章将简要介绍语义网及相关技术。探讨的话题包括关联数据、微格式和资源描述框架（resource description framework，RDF），还用示例演示了如何从 DBpedia 和维基百科中挖掘语义信息。

本书需要的工具

本书中的代码示例可以在 Linux、macOS 和 Windows 系统的较新版本 Python 中运行。代码已经在 Python 3.4.*和 Python 3.5.*中测试过，旧版本（Python 3.3.*或 Python 2.*）可能不支持。

第 1 章介绍了配置本地运行环境的步骤，以及本书会用到的工具。我们将使用与 Python 科学计算（如 NumPy、pandas 和 matplotlib）、机器学习（如 scikit-learn）、自然语言处理（如 NLTK）和社会网络分析（如 NetworkX）相关的一些基本程序库。

目标读者

本书面向希望使用公共 API 从社会媒体平台获取数据并进行统计分析，以从中获得有用信息的中级 Python 程序员。本书假设你对 Python 标准库有基本的了解，并提供了实用案例来引导你创建基于社交数据的分析项目。

排版约定

本书采用不同的文本样式来区分不同类别的信息。以下展示了部分样式的示例及其相应的含义。

正文中的代码、数据库表名和 Twitter 账号按以下样式显示："并且，这里的 `genre` 属性是一个元素数量可变的 Python 列表。"

代码块的样式如下所示：

```
from timeit import timeit
import numpy as np

if __name__ == '__main__':
  setup_sum = 'data = list(range(10000))'
  setup_np = 'import numpy as np;'
  setup_np += 'data_np = np.array(list(range(10000)))'
```

当想让你注意某部分代码时，我们会用粗体表示，如下所示。

```
Type your question, or type "exit" to quit.
> What's up with Gandalf and Frodo lately? They haven't been in the Shire
for a while...
Question: What's up with Gandalf and Frodo lately? They haven't been in the
Shire for a while...
Predicted labels: plot-explanation, the-lord-of-the-rings
```

命令行中的输入或输出内容的格式如下：

```
$ pip install --upgrade [package name]
```

新术语和关键词以黑体字显示。屏幕上（如菜单或对话框中）出现的文字按照如下样式显示："在**密钥与访问令牌**配置页面，开发者可以看到 API 密钥和密码，以及访问令牌和访问令牌密钥。"



此图标表示警告或重要事项。



此图标表示提示或小窍门。

读者反馈

我们期待读者的反馈。告诉我们你对本书的看法，喜欢什么或者不喜欢什么。读者反馈对我们很重要，因为这有助于我们出版充分满足读者需求的图书。要想提供反馈，只要发送电子邮件到 feedback@packtpub.com，并在邮件主题中注明书名即可。如果你擅长某一方面并且有兴趣撰写或者参与编写图书，可以在 http://www.packtpub.com/authors 中查看作者指南。

客户支持

为自己拥有一本 Packt 出版的图书而自豪吧！为了让你的书物有所值，我们还为你准备了以下内容。

下载示例代码

如果你是从 http://www.packtpub.com 网站购买的图书，登录自己的账号后就可以下载所有已购图书的示例代码。如果你是从其他地方购买的图书，请访问 http://www.packtpub.com/support 网站并注册，我们会将代码文件直接发送到你的电子邮箱。

你也可以通过以下步骤下载代码文件。

(1) 使用你的电子邮箱地址和密码在我们的网址上登录或注册。
(2) 将鼠标移到页面上方的 SUPPORT 标签。
(3) 点击 Code Downloads & Errata。
(4) 在 Search 框中输入书名。
(5) 选择你需要下载代码文件的图书。

(6) 在下拉菜单中选择你购买本书的途径。
(7) 点击 Code Download。

下载文件后，确保使用以下工具的最新版本来解压或提取文件夹：

- WinRAR / 7-Zip（Windows）
- Zipeg / iZip / UnRarX（Mac）
- 7-Zip / PeaZip（Linux）

本书代码也托管在 GitHub 上，访问 https://github.com/bonzanini/Book-SocialMediaMiningPython 即可获取。Packt 拥有丰富的图书和视频资源，GitHub 仓库 https://github.com/PacktPublishing/提供了这些资源的相关代码。欢迎查阅！

下载本书中的彩色图片

我们也为你提供了一份 PDF 文件，其中包含了本书中的截屏和图表等彩色图片。这些彩色图片可以帮助你更好地理解输出的变化。下载地址：https://www.packtpub.com/sites/default/files/downloads/MasteringSocialMediaMiningWithPython_ColorImages.pdf。

勘误

虽然我们竭力确保图书内容的正确性，但错误在所难免。如果你在我们出版的任何一本图书中发现了文本或代码错误，希望你能告知我们，我们将非常感激。你的善举足以减少其他读者在阅读出错内容时的纠结和不快，并帮助我们在后续版本中更正错误。如果你发现任何错误，请访问 http://www.packtpub.com/submit-errata，选择相应图书，点击 Errata Submission Form 链接，并输入错误之处的具体信息。[①]提交的错误得到验证后，我们就会接受你的建议，并将该处错误信息上传到我们的网站或添加到已有勘误表的相应位置。

访问 https://www.packtpub.com/books/content/support 并在搜索框中输入书名，即可查看该图书已有的勘误信息。这部分信息会在 Errata 部分显示。

反盗版

任何媒体都会面临版权内容在互联网上的盗版问题，Packt 也不例外。Packt 非常重视版权保护。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供相关网址或网站名称，以便我们进行补救。

① 本书中文版的勘误请到 http://www.ituring.com.cn/book/2005 查看和提交。——编者注

请将疑似盗版材料的网址发送到 copyright@packtpub.com。

你的反盗版行动就是在保护作者和出版社，只有这样，我们才能继续以优质内容回馈像你这样的热心读者。

问题

如果对本书存有任何方面的疑问，可以通过 questions@packtpub.com 邮箱联系我们，我们将尽力为你答疑解惑。

电子书

扫描如下二维码，即可购买本书电子版。



目　　录

第 1 章

社会媒体、社交数据和 Python

本书的主要内容是用 Python 将**数据挖掘**技术应用于**社会媒体**。这句话中用黑体突出的三个关键词帮助我们定义了本书的目标读者：对涉及上述三个主题的领域感兴趣的所有开发者、工程师、分析师、研究人员或学生。

本章包含如下主题：

- 社会媒体和社交数据
- 在社会媒体中进行数据挖掘的总体流程
- Python 开发环境的安装与配置
- Python 的数据科学工具
- 用 Python 处理数据

1.1 入门

2015 年第二季度，Facebook 宣称其已拥有近 15 亿月活跃用户。2013 年，Twitter 宣称其平台每天发布的推文超过 5 亿条。2015 年，你应该很感兴趣的、规模小一个数量级的 Stack Overflow 宣称，自网站运行以来，平台上已经积累了超过 1000 万个编程问题。

在描述社会媒体的流行程度时，这些数字仅仅是冰山一角。随着更多用户通过不同的平台分享越来越多的信息，社会媒体已经呈指数级增长。这些数据为数据挖掘者提供了难得的机会。本书的目的是指导你通过社会媒体 API 来收集数据，并用 Python 工具分析数据，最终获得有关用户如何在社会媒体上互动的有趣洞见。

本章将先介绍社会媒体挖掘的挑战和机遇，然后介绍后文会用到的一些 Python 工具。

1.2　社会媒体——机遇和挑战

在传统媒体时代，用户仅仅是消费者。信息流是单向的，即从信息发布者到用户。社会媒体打破了这种模式，它让每个用户既可以是消费者，同时也可以是发布者。很多学术著作都研究过这个话题，并借此来定义**社会媒体**这个术语。（例如，2010 年，Andreas M. Kaplan 和 Michael Haenlein 发表了学术论文“Users of the world, unite! The challenges and opportunities of Social Media”。）各种社会媒体平台都具有以下三个特征：

- 基于互联网的应用
- 用户生成的内容
- 网络

社会媒体是基于互联网的应用。显然，互联网和移动技术的发展促进了社会媒体的扩张。你可以用手机立刻连接到一个社会媒体平台，然后发布内容或了解最新的新闻。

社会媒体平台是由用户生成的内容驱动的。与传统媒体模式相反，每个用户都是潜在的内容发布者。更重要的是，任何用户都可以分享内容、发表评论，或者通过**喜欢**按钮（有时是赞同或点赞）表达积极的赞赏，从而与其他用户进行互动。

社会媒体是关于网络的。正如前面所描述的，社会媒体是关于用户与其他用户的互动。连接（connected）是大多数社会媒体平台的核心概念，通过**新闻订阅**和**时间线**所消费的内容是由你的连接所驱动的。

因为这些主要特征是多数社会媒体平台的核心，所以社会媒体有多种用途。

- 与朋友和家人保持联系（例如，通过 Facebook）
- 发微博和了解最新的新闻（例如，通过 Twitter）
- 与职业圈保持联系（例如，通过 LinkedIn）
- 分享多媒体内容（例如，通过 Instagram、YouTube、Vimeo 和 Flickr）
- 找到问题的答案（例如，通过 Stack Overflow、Stack Exchange 和 Quora）
- 找到并组织感兴趣的事物（例如，通过 Pinterest）

本书致力于回答一个核心问题：如何从社会媒体数据中提取有用的知识？退一步来看，我们首先需要定义什么是**知识**，以及什么是**有用**。

知识的传统定义来源于信息科学。知识的概念通常表示为金字塔的一部分，该金字塔有时也称作**知识层次**，其中数据是基础，信息是中间层，而知识在最顶层。知识层次如图 1-1 所示。

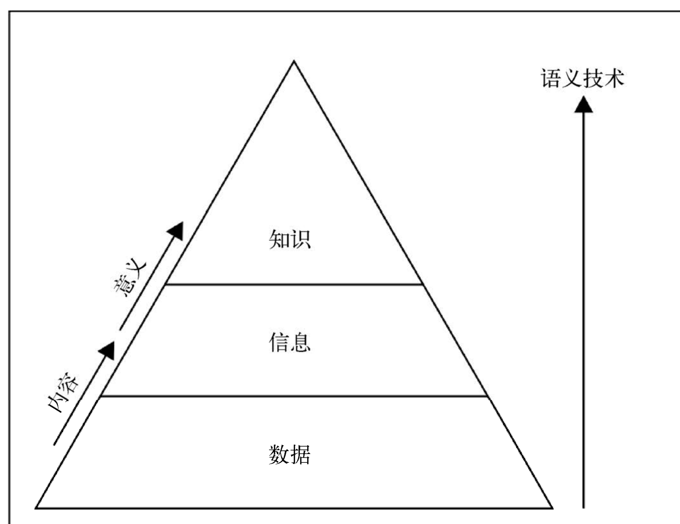


图 1-1 从原始数据到语义知识

攀登金字塔意味着从原始数据中提炼出知识，这需要经历内容和意义的整合。当攀登这个金字塔时，所用的技术可以帮助我们更深入地理解原始数据，更重要的是可以帮助我们理解生成这些数据的用户。换言之，它会变得更加有用。

在此背景下，有用的知识意味着**可执行的**知识，即能够使决策者实现一个商业战略的知识。本书将介绍从社交数据中抽取价值的关键原则。理解用户如何通过社会媒体平台进行互动也是本书的重要目标之一。

接下来将介绍从社会媒体平台挖掘数据的机遇和挑战。

1.2.1 机遇

开发数据挖掘系统的主要机遇是从数据中**获取有用的洞见**。其目的是用数据挖掘技术回答有意义的（有时是很难的）问题，从而帮助我们增长有关特定领域的知识。例如，在线零售商店可以应用数据挖掘来了解顾客的购物行为。通过分析数据，店主就可以基于顾客的购买习惯（例如，购买了 A 商品的用户一般还会购买 B 商品）向他们推荐产品。这样的推荐通常可以提供更好的用户体验，提升用户满意度，而回报则是更好的销售业绩。

不同领域的机构都可以用数据挖掘技术来改善业务。具体示例如下所示。

- 银行：
  - 识别忠诚客户，并为他们提供有针对性的促销方式

  - ■ 识别虚假交易的模式，以减少经济损失

- ❑ 医药：
  - ■ 理解病人的行为，以预测其诊疗时间
  - ■ 根据病人的历史数据支持医生的诊断决策

- ❑ 零售：
  - ■ 理解购物模式，以改善顾客体验
  - ■ 提高市场营销的精准性和有效性
  - ■ 分析实时的交通数据，以找到配送食物的最快路径

以上应用场景如何映射到社会媒体领域？解决该问题的核心在于用户如何通过社会媒体平台分享数据。很多机构不再局限于分析能够直接获取的数据，还会用社会媒体平台获取更多数据。

可以用设计优良、无关语言的 API 搜集社会媒体数据。社会媒体平台通常会为开发者提供一个 Web API，以便将他们的应用与特定的社会媒体功能整合起来。



API

API（application programming interface，应用编程接口）是一组操作定义和协议，描述了一个软件组件（如库或远程服务）的行为，其中包括它允许的操作、输入和输出。在使用第三方 API 时，开发者无须担心组件的内部实现，只需要关心如何使用该接口。

当谈到 Web API 时，我们指的是一种 Web 服务，该服务将多个 URI 暴露给公众以获取数据，这些 URI 可能暴露在鉴权层[①]后。一种常见的 API 架构方法称作**表现层状态转化**（representational state transfer，REST）。实现 REST 架构的 API 称作 **RESTful API**。我们仍然更喜欢 Web API 这个更常用的名称，因为很多现有的 API 并未严格遵守 REST 规则。阅读本书时，你并不需要深入理解 REST 架构。

1.2.2　挑战

社会媒体挖掘的部分挑战源自数据挖掘领域。

当处理社交数据时，我们常常需要处理**大数据**。为了理解大数据的含义及其中的挑战，我们需要回到大数据的传统定义（参见 Doug Laney 于 2001 年撰写的文章“3D Data Management: Controlling Data Volume, Velocity and Variety”）。该定义也称作**大数据的 3V**，即容量（volume）、多样性（variety）和速度（velocity）。经过这些年的发展，这个定义也在逐渐深化，人们在其基

① API 用户需要先注册账户，获取鉴权许可，然后才能使用 URI。——译者注

础上增加了更多的 V，其中最重要的是价值（value），因为探索大数据的一个主要目的是为机构提供价值。在大数据的原始 3V 中，**容量**指的是处理分布在不同机器中的数据。这就意味着需要一个与小数据（如内存数据）处理完全不同的基础架构。此外，容量与**速度**是相关的，当数据快速增长时，**大**这个概念也在不断变化。最后，**多样性**是指如何以不同的格式和结构呈现数据，这些多样的数据通常不兼容且带有不同语义。社会媒体数据也具备 3V 特征。

大数据的兴起促进了新的数据库技术 NoSQL 的发展。NoSQL 是多种数据库范式的总称，它们都脱离了传统的关系型数据，支持动态的模式设计。虽然本书并不是关于数据库技术的，但从数据库技术的角度来看，我们仍然重视对结构化数据、非结构化数据和半结构化数据的混合数据进行处理的需求。**结构化数据**指的是组织良好且通常以表格形式呈现的信息。因此，它与关系型数据库的关联就非常明显了。表 1-1 展示了一个结构化数据的示例，该数据表示一个书店销售的图书。

表 1-1 图书销售信息

书 名	类 型	价 格
《1984》	政治小说	12
《战争与和平》	战争小说	10

以上数据是结构化的，因为其中的每一个表示项都有精确的组织，即均具有书名、类型和价格这 3 个属性。

与结构化数据对立的是非结构化数据，后者是没有预定义数据模型的信息，或者说是没有根据预定义数据模型进行组织的信息。非结构化数据通常以文本数据的形式呈现，如电子邮件、文档、社会媒体帖子等。本书介绍的技术可用于从这些非结构化数据中抽取模式，从而提供结构化数据。

介于结构化数据和非结构化数据之间的是半结构化数据。半结构化数据的结构要么很灵活，要么是没有完全预定义的。有时半结构化数据也称作自我描述结构。一种典型的半结构化数据格式是 JSON。顾名思义，JSON 从编程语言 JavaScript 中借鉴了表示方法。因为广泛用于在 Web 应用中的服务端与客户端之间交换数据，所以 JSON 数据格式现在非常流行。以下是 JSON 格式数据的一个例子，它对表 1-1 中的图书数据进行了扩展。

```
[
  {
    "title": "1984",
    "price": 12,
    "author": "George Orwell",
    "genre": ["Political fiction", "Social science fiction"]
  },
  {
    "title": "War and Peace",
```

```
    "price": 10,
    "genre": ["Historical", Romance", "War novel"]
  }
]
```

从以上示例中可以观察到，第一本书有一个 author 属性，但该属性并未出现在第二本书中。另外，这里的 genre 属性是以一个列表的形式出现的，该列表具有不同数量的值。通常来说，在具有良好组织结构的（关系型）数据格式中应该避免这两方面，但它们可以在 JSON 中实现，而且在处理半结构化数据时更普遍。

对于结构化和非结构化数据的讨论其实是为了阐明如何处理不同的数据格式，并用不同的方法实现**数据完整性**。数据完整性主要用于应对脏数据、非一致性数据或不完整数据的组合挑战。

在分析用户生成的内容时，数据不一致和不完整是非常常见的，需要特别注意，尤其是来自社会媒体的数据。通常情况下，用户有条理地分享数据是非常罕见的。相反，社会媒体通常由非正式的环境以及一些自相矛盾的信息组成。例如，如果一个用户想要在某个公司的 Facebook 页面上抱怨该公司的某款产品，那么他首先得**喜欢**该页面，而这造成的效果与用户实际想要表达的结果相反，因为用户原本是想表达对该公司低质量产品的不满。因此，理解用户如何在社会媒体平台进行互动对设计良好的分析模型非常重要。

要创建数据挖掘应用，还需要考虑与**数据获取**相关的问题，特别是公司的策略是保护数据时。换句话说，数据并不总是能公开获得的。前文讨论过，与其他企业环境相比，在社会媒体挖掘中获取数据并不是困难的事情，因为大多数社会媒体平台都提供了设计良好且无关语言的 API，我们可以用它们来获得需要的数据。当然，这些数据的可获得性仍然取决于用户如何分享数据以及如何授权我们获取数据。例如，Facebook 用户可以设置其信息在公开资料中的公开程度，以及对好友公开的程度。生日、当前位置、工作经历等信息可以单独设置为保密或公开。同样，当我们试图通过 Facebook API 获取这些数据时，注册我们应用的用户可以只授权我们希望得到的部分数据。

数据挖掘的最后一个挑战在于理解数据挖掘过程本身以及对其进行解释。换句话说，在开始分析数据前，提出合适的问题并不总是很容易的。**研发**过程往往是由探索性分析驱动的，为了理解如何解决问题，我们需要先搞清楚问题。可以用**存在相关性并不意味着有因果关系**来描述统计学中的一个相关概念。很多统计检验可以用于建立两个变量间的相关关系，也就是说，两个事件一起发生，但这并不足以在任一方向建立因果关系。关于奇怪关联的有趣示例在网上随处可见。Franz H. Messerli 于 2012 年撰写的文章“Chocolate Consumption, Cognitive Function, and Nobel Laureates”是一个非常流行的示例，该文章发表于《新英格兰医学杂志》（最负盛名的医学杂志之一），展示了各国人均消耗的巧克力数量与诺贝尔奖数量间的相关关系。

当进行探索性分析时，非常重要的一点是，记住相关性（两个事件一起发生）是双向关系，而因果（事件 A 导致事件 B）是单向关系。是巧克力能使你变得更聪明，还是聪明人比普通人更

爱吃巧克力？这两个事件一起发生仅仅是偶然？是否存在第三个尚未发现的变量在这个相关关系中起作用？简单地观察相关性并不足以描述因果，但这通常是针对观察数据提出重要问题的一个有趣起点。

接下来将介绍我们的应用如何与社会媒体 API 交互，以及如何进行社会媒体数据分析。

1.2.3　社会媒体挖掘技术

本节将简要介绍构建一个社会媒体挖掘应用的整个流程，并在后续章节中深入介绍其中的细节。

整个流程分为以下步骤：

(1) 鉴权
(2) 数据收集
(3) 数据清洗和预处理
(4) 建模和分析
(5) 结果呈现

图 1-2 展现了该流程的概览。

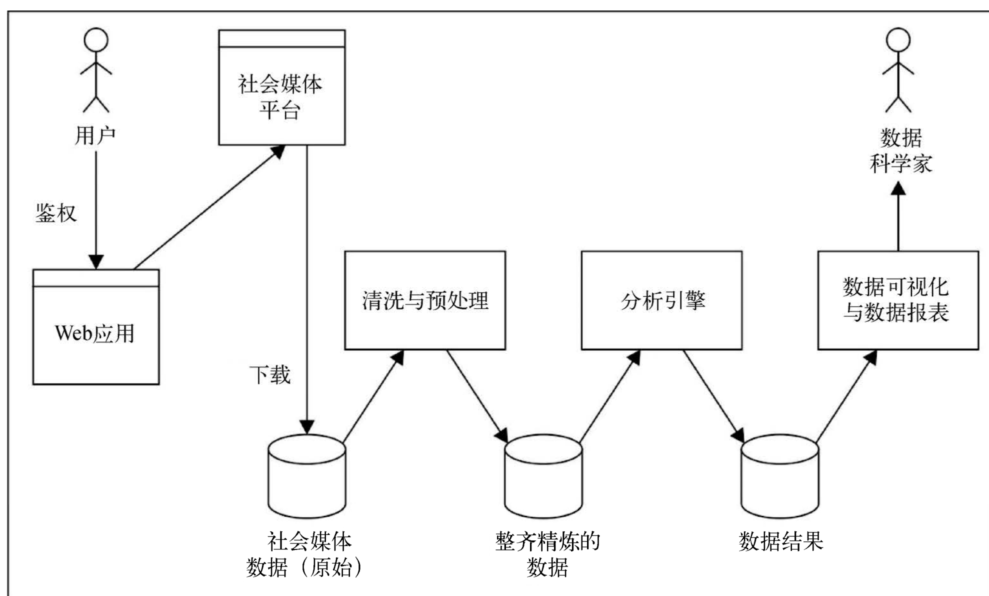


图 1-2　社会媒体挖掘的整个流程

通常用名为 OAuth（Open Authorization，开放授权）[①]的行业标准执行鉴权步骤。这个过程涉及三个角色：用户、消费者（我们的应用）和资源提供方（社会媒体平台）。该过程的步骤如下所示。

(1) 用户同意并授权第三方应用接入社会媒体平台。

(2) 用户并不是直接为第三方应用提供社会媒体的密码，资源提供方会生成一个令牌和一个密码，并将其交给第三方应用。第三方应用在每次请求时都需要使用这两个信息，以防伪造。

(3) 然后用户用该令牌重定向到资源提供方，该过程需要确认向第三方应用开放了获取用户数据的授权。

(4) 根据社会媒体平台的性质，还需要确认第三方应用能否代表用户来执行任意操作，如发布一个更新、分享一个链接等。

(5) 资源提供方为第三方应用发布一个有效的令牌。

(6) 然后该令牌可以返回给用户，以便用户确认接入权限。

图 1-3 展示了 OAuth 过程并标注了上述各个步骤。需要记住的是，交换凭证（用户名/密码）仅仅发生于步骤(3)和步骤(4)中的用户和资源提供方之间。所有其他交换都是由令牌驱动的。

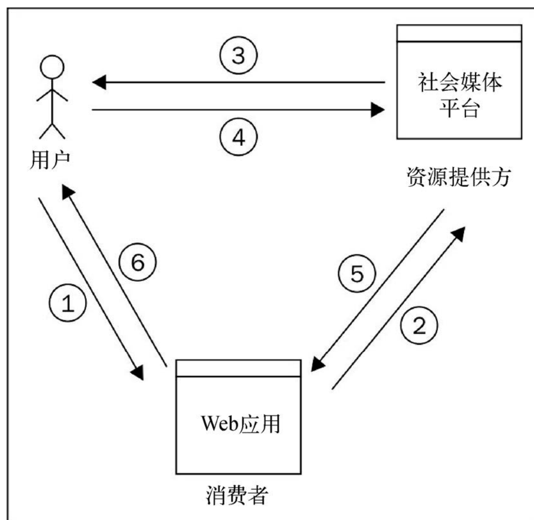


图 1-3　OAuth 过程

从用户的角度来看，当访问我们的 Web 应用并点击**用 Facebook（或 Twitter、Google+等）登录**这个按钮时，这个复杂的过程就会发生。然后用户必须确认他们要为我们的应用授权，并且所有过程都在底层完成。

---

① 如果想深入了解 OAuth，请关注人民邮电出版社即将出版的《OAuth 2 实战》（http://www.ituring.com.cn/book/2013）。——编者注

从开发者的角度来看，Python 生态系统的美妙之处在于，它为大多数社会媒体平台提供了非常成熟的库，其中就包含了鉴权过程的实现。作为开发者，一旦你为自己的应用注册了目标服务，平台将为其提供必要的授权令牌。图 1-4 展示了一个名为 Intro to Text Mining 的自定义 Twitter 应用的截图。在 Keys and Access Tokens 配置页面，开发者可以看到 API 密钥和密码，以及访问令牌和访问令牌密码。我们将在后续章节中讨论每个社会媒体平台的鉴权细节。

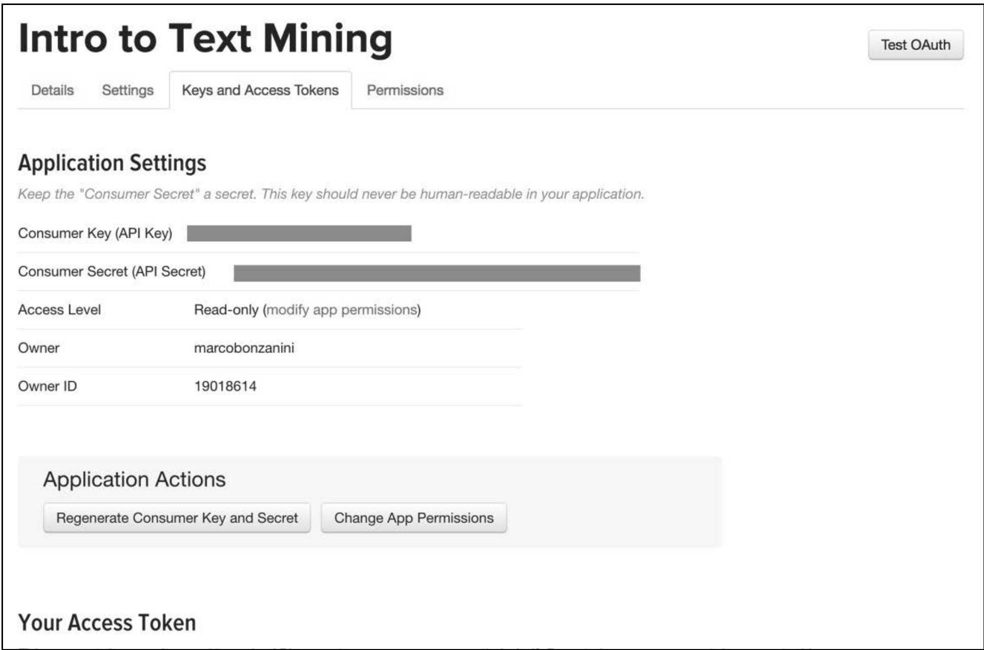


图 1-4　名为 Intro to Text Mining 的 Twitter 应用的配置页面，其中包含了开发者在其应用中使用的所有授权令牌

数据的收集、清洗和预处理步骤也依赖于社会媒体平台。特别是数据收集步骤和初始授权是捆绑在一起的，我们只能下载已授权获取的数据。另一方面，清洗和预处理为数据建模和分析做准备，以产生有关这些数据的洞见。

回到图 1-2，建模和分析是通过名为**分析引擎**的组件来执行的。本书中的典型数据处理任务就是文本挖掘和图挖掘。

文本挖掘（也称作文本分析）是从非结构化的文本数据中获得结构化信息的过程。文本挖掘适用于大多数社会媒体平台，因为用户可以用帖子或评论的方式发布内容。

以下列举了文本挖掘应用的一些示例。

- **文档分类**：将一个文档指定为一个或多个类别

- **文档聚类**：将文档分组为子集（也称作簇），每个类中的数据是一致的，且与其他类中的数据有区别（例如，通过话题或子话题进行区分）
- **文档摘要**：生成一个简短版文档，目的是为用户减少信息量，同时保留原始版本中最重要的内容
- **实体抽取**：在文本中定位并对实体分类，类别如人物、地点或机构等
- **情感分析**：识别并对文本中的情感和意见分类，以理解对特定产品、话题、服务等的态度

虽然并非所有这些应用都是为社会媒体量身定做的，但社会媒体平台上的文本数据的急剧增长使社会媒体成为了文本挖掘的研究对象。

图挖掘同样关注数据的结构。图是一种易于理解且强大的数据结构，可以用于不同的数据表示场景。在图中，需要考虑到两个主要成分：节点和边。前者表示实体或对象，后者表示节点间的关系或连接。在社会媒体中，图的显著用途是表示用户间的社会关系。从更通常的意义来说，在社会科学中用于表示社会关系的图也叫作社会网络。

在社会媒体中使用图这种数据结构时，我们可以自然地用节点表示用户，用边表示他们之间的关系（如**朋友**或**粉丝**）。这样，**朋友的朋友中喜欢 Python 的人**这种信息就很容易通过遍历图（即从一个节点出发，沿着所有的边行走）获得。图论和图挖掘为发掘更深层次的信息提供了可能，正如前面的示例所示，这些深层次的信息并不是那么显而易见。

介绍完社会媒体挖掘后，接下来将介绍一些经常用于数据挖掘项目的有用的 Python 工具。

1.3　Python 的数据科学工具

到目前为止，我们用数据挖掘这个术语来指我们将在本书中应用的问题和技术。实际上，本节的标题中提到了**数据科学**这个术语。近些年，这个术语的使用呈爆炸式增长，特别是在商业环境中，但很多学者和记者也抨击了将其作为流行语使用。同时，其他一些学术机构开始开设数据科学课程，很多图书和文章也关于这个主题。与其纠结于不同学科的边界应该在哪里，不如只观察人们的兴趣点为何如今普遍聚集于这些领域，包括数据科学、数据挖掘、数据分析、统计、机器学习、人工智能、数据可视化等。我们讨论的主题本质上就是跨学科的，它们之间随着时间的变化相互借鉴。毫无疑问，现在正是研究这些领域的最佳时机，因为公众对此有极大兴趣，并且有趣的项目在不断获得新的进展。

本节的目的是介绍 Python 的数据科学工具，以及后续章节中将用到的 Python 生态系统的部分内容。

Python 是适用于数据分析项目的最有趣的语言之一，以下是一些理由：

- 声明式和直观式语法

- 数据处理的丰富生态系统
- 高效性

优雅的语法使得 Python 的学习曲线比较平缓。作为一种动态的解释性语言，它推动了快速开发和交互式探索的发展。接下来将对数据处理的生态系统进行部分介绍，即本书会用到的主要的包。

从效率方面来说，解释性的高级语言并不以快速著称。像 NumPy 这样的工具在底层与低级库挂钩，并用友好的 Python 界面来实现较高的效率。此外，很多项目使用 Python 的超级子集 Cython，它通过允许定义强变量类型并用 C 语言编译来丰富 Python 语言。Python 世界中的很多其他项目正在尝试解决效率问题，以期使纯 Python 实现更加快速。本书不会深入介绍 Cython 或上述任何项目，我们只会用 NumPy（特别是通过使用了 NumPy 的其他库）来进行数据分析。

1.3.1　Python 开发环境的安装

当开始撰写本书时，Python 3.5 刚刚发布，其最新的一些功能受到了广泛关注，例如，进一步支持异步编程和类型提示的语义定义。就使用来说，Python 3.5 可能还没有普及，但是它代表了这门语言的当前发展状况。



本书中的示例与 Python 3 兼容，特别是 3.4+和 3.5+版本。

在无休止地讨论使用 Python 2 还是 Python 3 时，需要注意的一点是，对 Python 2 的支持将在几年后（在撰写本书时，该时间节点是 2020 年）消失。届时 Python 2 将不再支持新的功能，只会进行一些 bug 修复。另一方面，很多库仍然选择先为 Python 2 开发，然后再加上对 Python 3 的支持。因此，某些库可能会存在兼容性问题，但这些兼容性问题很快就会被社区解决。总之，如果没有强烈的理由影响选择，应该选择 Python 3，特别是对新项目来说。

1. `pip` 和 `virtualenv`

为了保持开发环境的整洁，并简化从原型到生产的转变，建议使用 `virtualenv` 来管理虚拟环境并安装依赖。`virtualenv` 是一个创建并管理独立 Python 环境的工具。通过使用一个独立的虚拟环境，开发者可以避免用互不兼容的库来污染全局 Python 环境。这些工具允许我们维护多个需要不同配置的项目，并轻松地进行项目的切换。更重要的是，虚拟环境可以安装在一个本地文件夹中，用户不需要管理者权限就可以使用。

为了在全局 Python 环境中安装 `virtualenv` 以供所有用户使用，可以在终端（Linux/Unix）或命令窗口（Windows）使用 `pip` 命令。

```
$ [sudo] pip install virtualenv
```

如果当前用户没有系统管理者权限，那么 Linux/Unix 或者 macOS 可能需要 sudo 命令。

如果一个包已经安装完毕，那么可以用以下命令将其升级到最新版本。

```
$ pip install --upgrade [package name]
```



自 Python 3.4 后，pip 工具已经集成在 Python 中了。之前的版本需要单独安装 pip，具体的安装方法参见其项目主页。也可以用以下命令将其升级到最新版本。

```
$ pip install --upgrade pip
```

当 virtualenv 全局可用后，对于每个项目，都可以定义一个单独的 Python 环境，其中依赖是独立安装的，不会干扰全局环境。这样，跟踪某个项目所需要的依赖就变得非常简单。

可以遵循以下步骤建立虚拟环境。

```
$ mkdir my_new_project # 创建一个新的项目文件夹
$ cd my_new_project # 进入项目文件夹
$ virtualenv my_env  # 安装自定义的虚拟环境
```

以上步骤会创建一个名为 my_env 的子文件夹，这也是我们在当前路径创建的虚拟环境的名称。在该子文件夹中，我们拥有创建独立 Python 环境所需的所有工具，其中包括 Python 二进制文件和标准库。为了激活该环境，可以输入以下命令。

```
$ source my_env/bin/activate
```

一旦环境被激活，命令窗口就会出现以下提示。

```
(my_env)$
```

可以用 pip 在该环境中安装 Python 包。

```
(my_env)$ pip install [package-name]
```

当环境处于激活状态时，使用 pip 安装的所有新 Python 库会安装到 my_env/lib/python{VERSION}/site-packages。注意，这是一个本地文件夹，因此不需要管理员权限就可以执行这个命令。

当想要退出虚拟环境时，可以使用以下简单命令。

```
$ deactivate
```

前面描述的过程适用于 Python 的官方发行版，你可以下载适用自己操作系统的相应版本。

2. Conda、Anaconda 和 Miniconda

此外，还可以考虑一个名为 Conda 的选项。因为它使包的依赖管理变得非常简单，所以在科学社区中备受关注。Conda 是一个开源的包管理器和环境管理器，可用于安装多种版本的软件

包（以及相关的依赖），这允许我们轻松地从一个版本切换到另一个版本。它支持 Linux、macOS 和 Windows。虽然最初是为 Python 创建的，但是 Conda 可用于任意软件的打包和分发。

Conda 现在主要有两个发行版：完备的 Anaconda 和轻量级的 Miniconda，其中前者装有约 100 个科学计算包，后者仅带有 Python 和 Conda 安装器，不带外部库。

如果你是 Python 新手，有时间下载较大的安装文件，拥有足够的磁盘空间，并且不愿意手动安装所有包，那么可以从 Anaconda 开始。对于 Windows 和 macOS 来说，可以使用图形界面或命令行安装器安装 Anaconda。图 1-5 展示了 macOS 上 Anaconda 安装步骤的截图。对于 Linux，只能使用命令行安装器。所有系统都可以选择安装 Python 2 或者 Python 3。如果希望全面掌控自己的系统，那么 Miniconda 将是你的最佳选择。

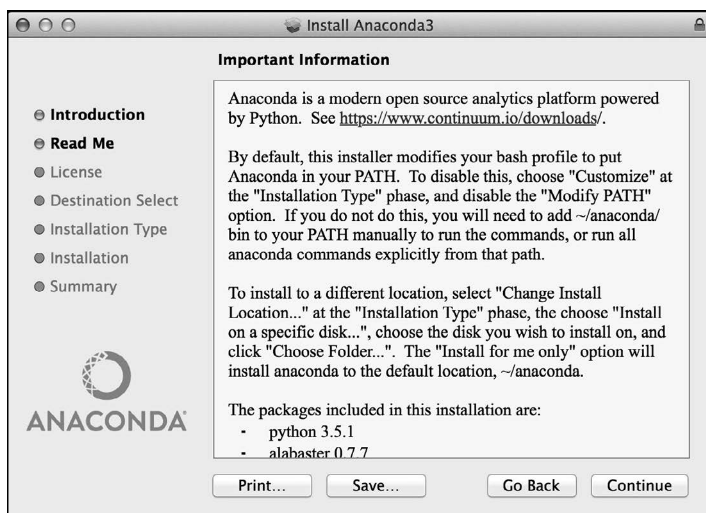


图 1-5　Anaconda 安装步骤的屏幕截图

安装了选择的 Conda 版本后，可以使用以下命令来创建一个新的 Conda 环境。

```
$ conda create --name my_env python=3.4 # 或者你更喜欢的版本
```

用以下命令激活环境。

```
$ conda activate my_env
```

与 `virtualenv` 类似，窗口中会提示环境的名称。

```
(my_env)$
```

可以用以下命令为这个环境安装新的包。

```
$ conda install [package-name]
```

最后，可以通过以下命令退出环境。

```
$ conda deactivate
```

Conda 的另一个良好特性是也支持用 pip 来安装包。因此，如果无法通过 conda install 获取一个库，或者该库还没有更新到最新版本，那么我们可以总是回退到传统的 Python 包管理器，同时使用 Conda 环境。

如果未特别指定，Conda 默认会在 https://anaconda.org 中查找包，而 pip 会用 PyPI（Python Package Index，也称作 CheeseShop）在 https://pypi.python.org/pypi 中查找。这两种安装器都可以指定从本地文件系统或私有仓库中安装包。

接下来我们将用 pip 安装所需要的包，但如果习惯使用 Conda，你也可以很容易地切换成 Conda。

1.3.2 高效的数据分析

本节会介绍用于 Python 科学计算的两个基础包：NumPy 和 pandas。

NumPy（数值 Python）提供了快速且高效的处理或类似于数组的数据结构。对于数值数据来说，用 Python 内置的数据结构（如列表或字典）进行存储和操作比用 NumPy 数组要慢得多。此外，其他库经常将 NumPy 数组当作输入或输出不同算法的容器，这些算法需要进行向量化操作。

为了用 pip 或者 virtualenv 安装 NumPy，可以使用以下命令。

```
$ pip install numpy
```

当使用完备的 Anaconda 发行版时，开发者会发现 NumPy 和 pandas 都已经预先安装好了，因此不需要上述安装步骤。

这个库的核心数据结构是名为 ndarray 的多维数组。

在交互式编译器中运行时，以下代码展示了如何用 NumPy 生成一个简单的数组。

```
>>> import numpy as np
>>> data = [1, 2, 3] # 一个整型列表
>>> my_arr = np.array(data)
>>> my_arr
array([1, 2, 3])
>>> my_arr.shape
(3,)
>>> my_arr.dtype
dtype('int64')
>>> my_arr.ndim
1
```

该示例表明，我们的数据由一个包含 3 个元素的一维数组（ndim 属性）表示。数组的数据类型是 int64，因为输入的所有数据都是整型。

通过用 timeit 模块来分析一个简单的操作（如列表的求和），我们可以观察 NumPy 数组的速度。

```
# Chap01/demo_numpy.py
from timeit import timeit
import numpy as np

if __name__ == '__main__':
  setup_sum = 'data = list(range(10000))'
  setup_np = 'import numpy as np;'
  setup_np += 'data_np = np.array(list(range(10000)))'

  run_sum = 'result = sum(data)'
  run_np = 'result = np.sum(data_np)'

  time_sum = timeit(run_sum, setup=setup_sum, number=10000)
  time_np = timeit(run_np, setup=setup_np, number=10000)

  print("Time for built-in sum(): {}".format(time_sum))
  print("Time for np.sum(): {}".format(time_np))
```

timeit 模块将一段代码作为第一个参数，并多次执行这段代码，然后将代码的运行时间作为输出。为了专注于我们想要分析的代码段，在 setup 参数中设置初始的数据准备和需要的包的导入，这些过程只会执行一次且不会计算在统计时间内。最后的参数 number 限制了迭代的次数为 10 000 次，而不是默认的 1 000 000 次。你观察到的输出应该如下所示。

```
Time for built-in sum(): 0.9970562970265746
Time for np.sum(): 0.07551316602621228
```

内置的 sum() 函数比 NumPy 的 sum() 函数慢 10 倍。对于更复杂的代码，我们也可以观察到很明显的数量级差异。



命名惯例

Python 社区汇集了一些事实标准来导入一些流行的库。NumPy 和 pandas 就是两个著名的示例，因为通常用别名导入它们，如下所示。

```
import numpy as np
```

这样一来，前面示例中的 NumPy 函数可以以 np.function_name() 的方式使用。类似地，pandas 库的别名是 pd。原则上，用 from numpy import * 导入库的所有命名空间是一种非常不好的做法，因为这会污染当前的命名空间。

以下是我们需要记住的 NumPy 数组的一些特征。

- NumPy 数组的大小在创建时就固定了，不像 Python 列表那样可以动态改变。因此，改变数组大小的操作实际上是创建一个新的数组并删除原来的数组

- 数组中每个元素的数据类型必须一致（由对象组成的数组例外，因此可能导致不同的内存大小）
- NumPy 鼓励使用向量操作，这样代码会更紧凑易读

本节要介绍的第二个库是 pandas。因为建立在 NumPy 之上，所以它也提供了非常快速的计算，以及名为 Series 和 DataFrame 的便利数据结构，允许我们灵活并准确地操作数据。

以下是 pandas 中非常好的一些功能：

- 实现数据操作的快速且高效的对象
- 读写不同格式（CSV、文本文件、MS Excel 表格或 SQL 数据结构）数据的工具
- 智能地处理缺失数据及相关的数据对齐
- 对大数据集进行基于标签的切片和分段
- 类似 SQL 的数据聚合和数据转换
- 支持时间序列功能
- 整合了画图功能

可以用以下命令从 CheeseShop 安装 pandas。

```
$ pip install pandas
```

以下是一个在 Python 交互式编译器中运行的示例，其中使用的是编造出来的用户数据。

```
>>> import pandas as pd
>>> data = {'user_id': [1, 2, 3, 4], 'age': [25, 35, 31, 19]}
>>> frame = pd.DataFrame(data, columns=['user_id', 'age'])
>>> frame.head()
   user_id  age
0        1   25
1        2   35
2        3   31
3        4   19
```

最初的数据布局是基于字典的，其中键是用户的属性（用户 ID 和年纪）。字典中的值是用列表表示的，对于每个用户来说，相应的属性是按照位置对齐的。一旦用这些数据创建 DataFrame，数据的对齐格式会立刻变得非常清楚。`head()` 函数能够以表格的形式打印出数据，并且当数据过大时，它只会打印出前 10 行。

现在我们在 DataFrame 中添加一列。

```
>>> frame['over_thirty'] = frame['age'] > 30
>>> frame.head()
   user_id  age over_thirty
0        1   25       False
1        2   35        True
2        3   31        True
3        4   19       False
```

如果使用 pandas 的声明式语法，就不需要迭代所有列来获得相应的数据，而是如前面的示例所示，可以使用一个类似于 SQL 的操作。这个操作用现有数据来创建一个由布尔值组成的列。还可以按照以下方式添加新列。

```
>>> frame['likes_python'] = pd.Series([True, False, True, True],
index=frame.index)
>>> frame.head()
   user_id  age over_thirty likes_python
0        1   25       False         True
1        2   35        True        False
2        3   31        True         True
3        4   19       False         True
```

可以用 describe()方法来观察一些基本的描述性统计结果。

```
>>> frame.describe()
        user_id   age over_thirty likes_python
count  4.000000   4.0           4            4
mean   2.500000  27.5         0.5         0.75
std    1.290994   7.0     0.57735          0.5
min    1.000000  19.0       False        False
25%    1.750000  23.5           0         0.75
50%    2.500000  28.0         0.5            1
75%    3.250000  32.0           1            1
max    4.000000  35.0        True         True
```

如以上结果所示，50%的用户超过 30 岁，75%的人喜欢 Python。



下载示例代码

前言介绍了下载代码的详细步骤。

GitHub 中也有本书的代码：https://github.com/bonzanini/Book-SocialMediaMining-Python。还可以在 https://github.com/PacktPublishing/中找到其他 Packt 图书的代码和视频。

1.3.3 机器学习

机器学习是构建算法以从数据中学习并进行预测的学科。它和数据挖掘紧密相关，有时这两个领域的概念可以互换。这两个领域的主要区别在于：机器学习侧重于基于数据的已知属性进行预测，而数据挖掘则侧重于基于数据的未知属性发现新的信息。两个领域会相互借鉴算法和技术。本书的目的之一是实用，因此，虽然这两个领域在学术上有很多重合，也有各自的目标和假设，但本书不会纠结于这些细节。

以下是机器学习的几个应用示例。

- 判断收到的电子邮件是否为垃圾邮件
- 从已知的主题（如体育、经济或政治）中为一篇新文章选择话题

- 分析银行交易数据，从中识别诈骗
- 对于 apple 这个词的查询，判断用户的意图是查找水果还是查找计算机

机器学习中最流行的一些方法可以分为监督学习和无监督学习两类，接下来会具体介绍。当然，这种简单的划分并不能代表机器学习领域的广度和深度，但这是我们了解其术语的一个良好起点。

监督学习可以用来解决分类这样的问题。在分类问题中，数据带有额外的属性，而我们想要预测类的标签。在这种情况下，分类器会将每个输入对象与期望输出关联起来。然后分类器基于输入对象的特征进行推断，为新的未见输入预测期望标签。监督学习的常用方法有**朴素贝叶斯**（naive Bayes，NB）、**支持向量机**（support vector machine，SVM）和**神经网络**（neural network，NN）系列模型，如感知器或多层感知器。

学习算法用来构建数学模型的样本输入称作**训练数据**，而我们想要预测的未见输入称为**测试数据**。机器学习算法的输入通常是向量形式的，向量的每个元素表示输入的一个**特征**。在监督学习中，指定给每个未见输入的期望输出通常称作**标签**或**目标**。

无监督学习应用于已知数据并不带有标签的问题。这类问题的一个典型示例就是聚类。在聚类问题中，算法会试图寻找数据中的隐藏结构，以便将相似的项分为一类。另一个应用是识别不属于特定组的项（如异常点检测）。最常用的聚类算法是 k-means。

用于机器学习的主要 Python 包是 scikit-learn，它是一个开源的机器学习算法集合，其中包括了获取和预处理数据、评估算法的输出，以及对结果进行可视化的工具。

你可以用以下步骤从 CheeseShop 下载 scikit-learn。

```
$ pip install scikit-learn
```

本书不会深入介绍这些机器学习方法，而是直接用 scikit-learn 来解决一个聚类问题。

到目前为止，我们还没有社交数据，因此可以使用 scikit-learn 提供的一个数据集。

我们将使用 Fisher 的鸢尾花数据集，它是 Ronald Fisher 在 20 世纪 30 年代创造的，并且是目前最经典的数据集之一：由于数据量较少，它通常在文献中用作简单的示例数据。鸢尾花数据集中共有 3 种鸢尾花，每种花有 50 个示例数据，而数据由 4 个特征组成：花瓣的长度、花瓣的宽度、花萼的长度和花萼的宽度。

鸢尾花数据集中的每个示例数据都带有正确的标签，所以常用作分类问题的示例，而比较少用于聚类问题，主要是因为只有两个有明显区别的簇。因为鸢尾花数据集较小且结构简单，所以我们用它来介绍如何使用 scikit-learn 进行数据分析。如果想要运行示例代码，包括数据可视化部分，那么还需要用 `pip install matplotlib` 来安装 matplotlib 库。本章后面还会更详细地介绍数据可视化。

先查看以下示例代码。

```
# Chap01/demo_sklearn.py
from sklearn import datasets
from sklearn.cluster import KMeans
import matplotlib.pyplotas plt

if __name__ == '__main__':
  # 导入数据
  iris = datasets.load_iris()
  X = iris.data
  petal_length = X[:, 2]
  petal_width = X[:, 3]
  true_labels = iris.target
  # 应用 k-means 聚类
  estimator = KMeans(n_clusters=3)
  estimator.fit(X)
  predicted_labels = estimator.labels_
  # 颜色定义方案：红、黄、蓝
  color_scheme = ['r', 'y', 'b']
    # 标记定义：圆圈、叉和加号
  marker_list = ['o', 'x', '+']
    # 将颜色/标记指定给预测标签
  colors_predicted_labels = [color_scheme[lab] for lab in
                             predicted_labels]
  markers_predicted = [marker_list[lab] for lab in
                       predicted_labels]
  # 将颜色/标记指定给真实标签
  colors_true_labels = [color_scheme[lab] for lab in true_labels]
  markers_true = [marker_list[lab] for lab in true_labels]
  # 绘制并保存两幅散点图
  for x, y, c, m in zip(petal_width,
                        petal_length,
                        colors_predicted_labels,
                        markers_predicted):
    plt.scatter(x, y, c=c, marker=m)
  plt.savefig('iris_clusters.png')
  for x, y, c, m in zip(petal_width,
                        petal_length,
                        colors_true_labels,
                        markers_true):
    plt.scatter(x, y, c=c, marker=m)
  plt.savefig('iris_true_labels.png')

  print(iris.target_names)
```

首先，我们将数据集导入 `iris` 变量，该变量是一个包含数据及其信息的对象。具体来说，`iris.data` 包含数据本身，数据的形式是一个 NumPy 数组或普通数组，而 `iris.target` 包含一个数值标签，该标签表示该数据所属的类。在每个示例向量中，其中的 4 个值分别表示花萼的长度（单位：厘米）、花萼的宽度（单位：厘米）、花瓣的长度（单位：厘米）、花瓣的宽度（单位：厘米）。利用 NumPy 数组的切片功能，可以抽取出第 3 个和第 4 个元素，然后分别赋给

petal_length 和 petal_width 变量。我们将用这两个变量画出示例数据的二维表示，尽管向量是四维的。

聚类过程仅仅包含两行代码：一行用于创建 KMeans 算法的一个实例，另一行用 fit() 函数对数据使用聚类模型。这种简洁的接口是 scikit-learn 的特征之一，它让你在大多数情况下只使用几行代码就能应用一个学习算法。对于 k-means 算法的应用，我们选择簇的数量为 3。需要注意的是，在实际运用中很难提前确定簇的数量。确定正确的簇数量本身就是一个挑战，并且取决于聚类算法本身。这里列举示例是为了简单介绍 scikit-learn 及其接口的简单性，因此简化了簇数量的确定方式。通常情况下，大量的工作都在于将数据处理为 scikit-learn 可以理解的格式。

上述示例中第二个部分的目的是用 matplotlib 对数据进行可视化。我们先利用在 color_scheme 列表中定义的红、黄和蓝三种颜色，定义一种颜色方案来区分三个簇。接下来，因为真实的标签和聚类结果是以从 0 开始的整数表示的，所以它们可用作索引，与其中一种颜色匹配。

注意，虽然真实标签的数量与标签的特定含义（即类名）相关，但簇的数量仅用于表明数据属于某一个簇，并不包含有关簇含义的相关信息。在本例中，真实标签的三个类是 setosa、versicolor 和 virginica，分别代表数据集中鸢尾花的三个品种。

示例中的最后几行代码生成了两幅散点图：一个是真实的标签，另一个是聚类结果。两图中用花瓣的长度和宽度作为 x 轴和 y 轴，如图 1-6 所示。在两幅图中，项的位置当然是相同的，但我们可以观察到算法如何将数据分割为 3 个组。具体来说，左下方的簇明显区别于其他两个簇，毫无疑问，算法可以很轻松地将其识别出来。而其他两个簇则更难区分，因为部分元素是重合的，所以算法存在误分类。

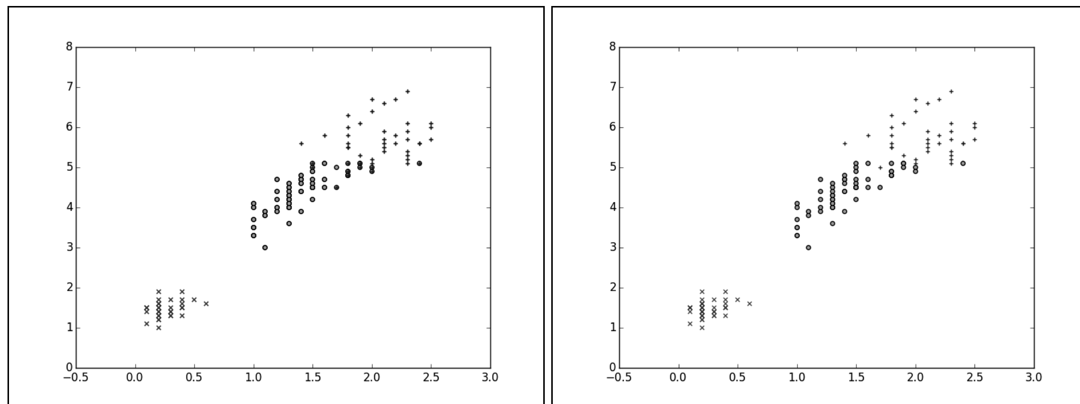


图 1-6 鸢尾花数据的二维表示，根据真实标签（左）和聚类结果（右）着色

需要再次强调的是，之所以能发现误分类，是因为我们知道每个数据的真实类。该算法基于输入数据的特征创建了一个输入与输出的相关关系。

1.3.4 自然语言处理

自然语言处理是一门研究有关自动分析、理解和生成自然语言的方法和技术的学科，其中自然语言包括人类自然书写和口述的语言。

从学术的角度来看，自然语言处理是一个活跃了很多年的领域，其早期的发展归功于计算机科学之父之一的艾伦·图灵。他在 1950 年提出了一个测试来评估机器的智力，理念非常直白：如果一个人类裁判与两个代理（分别是机器和人类）进行书面会话，那么机器能让裁判认为其不是一台机器吗？如果能，那么机器就通过了测试，表明它具备了**智能**。

这个测试就是大家今天熟知的图灵测试，它在很长一段时间内仅仅是计算机科学领域的常识，而现在已通过大众媒体为更多人所知。2014 年上映的电影《模仿游戏》就是基于艾伦·图灵的生平改编的，片名也清楚地指代了图灵测试本身。与图灵测试相关的另一部电影是 2015 年上映的《机械姬》，该片强调了人工智能的发展可以使得其为了自身利益而欺骗并愚弄人类。片中，人类裁判和人形机器人 Ava 通过直接的言语交互进行了图灵测试。这里我们不会向还未看过这部电影的人剧透结果，但影片中人工智能以阴暗狡猾的方式变得越来越聪明，而且比人类更聪明。有趣的是，未来的机器人是通过搜索引擎日志进行训练的，以便其理解和模仿人类提问的方式。

介绍人工智能的历史和假想未来是为了强调掌握人类语言对于高级人工智能发展至关重要。尽管近几年有较大发展，但人工智能仍然没有达到掌握人类语言的终极目标，自然语言处理仍然是当前的一个热门主题。

在社会媒体的竞争中，大量的自然语言等待挖掘，这对我们来说就意味着机会。社会媒体上的文本数据在持续增长，很多问题或许已经有了答案，但将原始文本转换为信息仍然不是一个简单的任务。社会媒体上时刻都在产生对话，人们在网络论坛上提出技术问题并查找答案，用户通过评论描述他们使用某款产品的体验。了解这些对话的话题、找到回答问题的专家用户、理解撰写评论的顾客的意见，这些任务都可以通过自然语言处理实现，并且达到一定的准确度。

回到 Python，其中最流行的自然语言处理包是 **NLTK**（Natural Language Toolkit，自然语言工具包）。该工具包为很多常见的自然语言处理任务提供了一个友好的界面，以及词汇资源和语言学数据。

我们可以用 NLTK 轻松完成一些任务，如下所示。

- 对单词和句子进行分词，即将文本分解为单个词项
- 对单词进行词性标注，即识别单词的句法功能，如名词、副词、动词等
- 识别命名实体，例如，识别并对人物、地点、机构实体分类
- 将机器学习技术（如分类）应用于文本
- 从原始文本中抽取信息

可以用以下步骤从 CheeseShop 安装 NLTK。

```
$ pip install nltk
```

NLTK 与其他包的不同之处在于，该框架中包含了针对特定任务的语言资源（即数据）。因为尺寸较大，所以这部分数据并没有包括在默认安装中，而是需要单独下载。

http://www.nltk.org/data.html 中包含详细的安装步骤，强烈建议你阅读该官方文档。

可以在 Python 编译器中输入以下代码。

```
>>> import nltk
>>> nltk.download()
```

如果是在桌面环境中，上述代码运行后会打开一个新的窗口，你可以在该窗口中浏览可用的数据。如果桌面环境不可用，你会在终端中看到一个文本界面。你可以选择下载单独的包，也可以下载全部数据（这需要约 2.2GB 的磁盘空间）。

如果你是在管理员账号下进行操作，该下载器会将文件存储在一个中央位置（Windows 中的 C:\nltk_data，Unix 和 Mac 中的/usr/share/nltk_data）。如果你是普通用户，下载器会将数据存储在主文件夹（如~/nltk_data）中。也可以自定义路径，但这样你就需要将路径写入`$NLTK_DATA` 环境变量，因为 NLTK 会查看该变量来确定到哪里查找需要的数据。

如果磁盘空间足够的话，安装所有的数据可能是最方便的选项，因为以后就再也不用考虑这件事了。另一方面，下载所有东西并不利于你搞清楚自己将需要哪些资源。如果希望全面控制自己安装的东西，可以一个一个地下载包。这样一来，应用 NLTK 时可能会碰到 `LookupError`，这意味着你缺失一些数据，并且必须下载。

例如，在最初安装 NLTK 后，如果试图在 Python 解释器中对一些文本进行分词，可以输入以下代码。

```
>>> from nltk import word_tokenize
>>> word_tokenize('Some sample text')
Traceback (most recent call last):
  # 很长的错误追踪
LookupError:
**********************************************************************
  Resource 'tokenizers/punkt/PY3/english.pickle' not found.
  Please use the NLTK Downloader to obtain the resource: >>>
  nltk.download()
  Searched in:
    - '/Users/marcob/nltk_data'
    - '/usr/share/nltk_data'
    - '/usr/local/share/nltk_data'
    - '/usr/lib/nltk_data'
    - '/usr/local/lib/nltk_data'
    - ''
**********************************************************************
```

这个错误提示我们，所有的常规文件夹中都未找到用于分词的 punkt 资源，因此，我们得回到 NLTK 下载器来解决这个问题。

假设现在安装了 NLTK 的全部数据，然后返回前面的示例，并详细讨论分词。

在自然语言处理中，分词（也称作切分）是将一段文本分解为更小单元（称作词项或段）的过程。尽管词项可以以很多不同的形式解释，但通常我们更关注单词和句子。使用 word_tokenize() 的一个简单示例如下所示。

```
>>> from nltk import word_tokenize
>>> text = "The quick brown fox jumped over the lazy dog"
>>> words = word_tokenize(text)
>>> print(words)
# ['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

word_tokenize()的输出是一个字符串列表，其中每个字符串表示一个单词。在这个示例中，单词的边界是由空白给出的。同样，sent_tokenize()也返回一个字符串列表，其中每个字符串表示一个句子，通过标点符号分割。以下示例用到了这两个函数。

```
>>> from nltk import word_tokenize, sent_tokenize
>>> text = "The quick brown fox jumped! Where? Over the lazy dog."
>>> sentences = sent_tokenize(text)
>>> print(sentences)
#['The quick brown fox jumped!', 'Where?', 'Over the lazy dog.']
>>> for sentence in sentences:
...     words = word_tokenize(sentence)
...     print(words)
# ['The', 'quick', 'brown', 'fox', 'jumped', '!']
# ['Where', '?']
# ['Over', 'the', 'lazy', 'dog', '.']
```

可以看到，标点符号本身也被视为词项，包含在 word_tokenize()的输出中。这提出了一个新的问题：如何定义一个词项？word_tokenize()函数实现的算法是为标准英语设计的，而本书关注的重点是社会媒体数据，那就需要探讨一下标准英语的规则是否同样适用于社会媒体内容。我们以虚构的 Twitter 数据为例，如下所示。

```
>>> tweet = '@marcobonzanini: an example! :D http://example.com #NLP'
>>> print(word_tokenize(tweet))
# ['@', 'marcobonzanini', ':', 'an', 'example', '!', ':', 'D', 'http', ':',
'//example.com', '#', 'NLP']
```

以上示例中的 Twitter 数据包含一些特殊符号，破坏了标准的分词。

- 用户名前面带有一个@符号，因此@marcobonzanini 被分为两个词项，其中@被当作标点符号
- 表情符号（如:D）在聊天、短信和社会媒体中很常用，但是并不属于标准英语，因此也被分割了

- URL 常用于分享文章或图片，但同样不属于标准英语，因此也被分解成了组成部分
- 散列标签（如#NLP）是指带有#前缀的字符串，用于定义帖子的主题，以便其他用户可以很轻松地搜索特定主题或跟踪对话

前面的示例展示了像分词这样明显的问题会隐藏很多难以处理的情况，因此需要比直觉更聪明的方法来解决问题。好在 NLTK 提供了以下解决方案。

```
>>> from nltk.tokenize import TweetTokenizer
>>> tokenizer = TwitterTokenizer()
>>> tweet = '@marcobonzanini: an example! :D http://example.com #NLP'
>>> print(tokenizer.tokenize(tweet))
# ['@marcobonzanini', ':', 'an', 'example', '!', ':D',
'http://example.com', '#NLP']
```

这个示例也展示了 NLTK 的简单接口。我们将在本书的不同场景中使用该框架。

随着自然语言处理应用的热度持续增长，Python-for-NLP 生态系统近几年来也在疯狂扩张，很多有趣的项目获得了越来越多的关注。举例来说，被称为**为人类设计的主题模型**的 Gensim 是一个开源库，主要注重于语义分析。Gensim 和 NLTK 一样提供了非常好用的接口，因此被加上了**为人类**的前缀。Gensim 这么流行的另一个原因是高效。它是一个为速度高度优化过的包，拥有分布式计算的选项，并且可以在不将所有数据放入内存的情况下处理大数据集。

可以用以下代码安装 Gensim。

```
$ pip install gensim
```

Gensim 主要依赖于 NumPy 和 SciPy。如果需要使用 Gensim 的分布式计算功能，你还需要安装 Python Remote Objects（Pyro4）。

```
$ pip install Pyro4
```

为了展示 Gensim 的简单接口，我们可以看看文本摘要模块。

```
# Chap01/demo_gensim.py
from gensim.summarization import summarize
import sys

fname = sys.argv[1]

with open(fname, 'r') as f:
  content = f.read()
  summary = summarize(content, split=True, word_count=100)
  for i, sentence in enumerate(summary):
    print("%d) %s" % (i+1, sentence))
```

demo_gensim.py 脚本需要一个命令行参数，即需要进行摘要的文本文件的名称。为了测试该脚本，我从维基百科的《魔戒》页面中摘了一段文本，即第一册《魔戒首部曲 · 护戒同盟》中的情节。用以下命令运行该脚本。

```
$ python demo_gensim.py lord_of_the_rings.txt
```

输出结果如下所示。

```
1) They nearly encounter the Nazgûl while still in the Shire, but shake off
pursuit by cutting through the Old Forest, where they are aided by the
enigmatic Tom Bombadil, who alone is unaffected by the Ring's corrupting
influence.
2) Aragorn leads the hobbits toward the Elven refuge of Rivendell, while
Frodo gradually succumbs to the wound.
3) The Council of Elrond reveals much significant history about Sauron and
the Ring, as well as the news that Sauron has corrupted Gandalf's fellow
wizard, Saruman.
4) Frodo volunteers to take on this daunting task, and a "Fellowship of the
Ring" is formed to aid him: Sam, Merry, Pippin, Aragorn, Gandalf, Gimli the
Dwarf, Legolas the Elf, and the Man Boromir, son of the Ruling Steward
Denethor of the realm of Gondor.
```

Gensim 中的 `summarize()` 函数实现了经典的 TextRank 算法。该算法根据句子的重要程度排序，并选择最特殊的句子来生成输出摘要。这种方法是**抽取式摘要**技术，即输出中只包含从输入中挑选出的句子，不经过任何文本变换、变形等。输出的文本大小约是原始文本的 25%。也可以通过 `ratio` 参数来设定这个比率，或者通过 `word_count` 指定固定数目的单词。在以上两种情况下，输出都将包含完整的句子，也就是说，句子不会因为兼顾输出的大小而被截断。

1.3.5 社会网络分析

网络理论是图论的一部分，是对表示离散对象间的关系的图的研究。它在社会媒体中的应用称作**社会网络分析**（social network analysis，SNA），主要研究社会结构，如朋友关系或相识关系。

NetworkX 是 Python 用于生成、操作和学习复杂网络结构的主要库之一。它提供了图的数据结构，以及很多著名的标准图算法。

可以用以下代码从 CheeseShop 安装 NetworkX。

```
$ pip install networkx
```

以下示例展示了如何用几个节点创建一个简单的图，图中的节点表示用户，节点间的连线表示用户间的社交关系。

```
# Chap01/demo_networkx.py
import networkx as nx
from datetime import datetime

if __name__ == '__main__':
  g = nx.Graph()
  g.add_node("John", {'name': 'John', 'age': 25})
  g.add_node("Peter", {'name': 'Peter', 'age': 35})
  g.add_node("Mary", {'name': 'Mary', 'age': 31})
```

```
  g.add_node("Lucy", {'name': 'Lucy', 'age': 19})

  g.add_edge("John", "Mary", {'type': 'friend', 'since': datetime.today()})
  g.add_edge("John", "Peter", {'type': 'friend', 'since': datetime(1990, 7, 30)})
  g.add_edge("Mary", "Lucy", {'type': 'friend', 'since': datetime(2010, 8, 10)})

  print(g.nodes())
  print(g.edges())
print(g.has_edge("Lucy", "Mary"))
# ['John', 'Peter', 'Mary', 'Lucy']
# [('John', 'Peter'), ('John', 'Mary'), ('Mary', 'Lucy')]
# True
```

所有的节点和边都能够以 Python 字典的形式加入额外的属性，这些属性可以用来描述网络的语义信息。

`Graph` 类用于表示无向图，即边是没有方向的。通过使用 `has_edge()` 函数来查看 Lucy 和 Mary 之间是否存在一条边，可以很清楚地看到这一点。在上例中，Lucy 和 Mary 之间确实存在一条边，但该函数也显示了其方向是被忽略的。节点到其自身的边也被忽略了，也就是说，只有一条边对应两个节点这种形式是有效的。`Graph` 类是可以使用自循环的，但我们的示例并不需要。

NetworkX 还支持有向图（`DiGraph`，节点间的方向很重要）以及节点间存在多条（平行）边的 `MultiGraph` 和 `MultiDiGraph`。

1.3.6 数据可视化

数据可视化是探索数据的视觉表示的一个交叉学科。视觉表示是一个强大的工具，可以帮助我们理解复杂的数据，以及有效地展示和交流数据分析过程的结果。通过对数据进行可视化，人们可以看到数据不那么显而易见的一面。如果说一幅图片包含千言万语才能传达的信息，那么良好的数据可视化可以让你借助一幅简单的图理解复杂的概念。例如，数据科学家在进行探索性数据分析时，可用数据可视化来帮助理解数据。此外，数据科学家也可以使用数据可视化与非专家进行交流，并向他们解释数据的精彩之处。

Python 提供了很多数据可视化工具，如 1.3.3 节中提到过的 matplotlib 库。可以用以下命令安装这个库。

```
$ pip install matplotlib
```

matplotlib 可以生成各种格式的出版质量的图像。这个库背后的思想是，开发者可以通过几行代码创建简单的图像。matplotlib 图像也可以存储为不同的文件格式，如 PNG 或 PDF。

我们在以下简单示例中画出了一些二维数据。

```
# Chap01/demo_matplotlib.py
import matplotlib.pyplot as plt
```

```
import numpy as np

if __name__ == '__main__':
  # 用红色的点画出 y = x^2
  x = np.array([1, 2, 3, 4, 5])
  y = x * x
  plt.plot(x, y, 'ro')
  plt.axis([0, 6, 0, 30])
  plt.savefig('demo_plot.png')
```

以上代码的输出如图 1-7 所示。

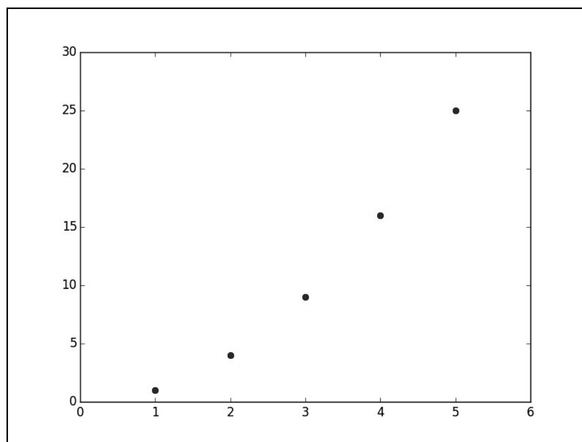


图 1-7　matplotlib 创建的图像

用别名 plt 指代 pyploy 是一个普遍的命名惯例。plot() 函数接受两个类似序列的参数，其中分别包含 x 和 y 坐标。在本例中，这两个坐标是用 NumPy 数组创建的，也可以用 Python 列表创建。axis() 函数定义了轴的显示范围。因为我们画的是 1~5 的平方，所以 x 轴的取值范围应该是 0~6，而 y 轴的取值范围应该是 0~30。最后，savefig() 函数创建了一个图像文件，如图 1-7 所示。我们可以由文件后缀猜到该图像的格式。

matplotlib 可以创建很棒的图片，但有时我们需要允许用户通过放大和缩小图像等方式的互动来探索数据。这种互动是其他编程语言擅长的领域。例如，JavaScript（特别是很流行的 D3.js 库）允许创建基于 Web 的交互式可视化。虽然这并不是本书的核心主题，但是我想指出 Python 在该领域并不落后，这得益于将 Python 对象翻译为 Vega 语法（基于 JSON 的声明式格式）的工具。因此，我们也可以用 Python 创建交互式可视化。

Python 和 JavaScript 可以良好合作的一个特别有趣的场景是地理数据的可视化。大多数的社会媒体平台是通过移动设备访问的，这就提供了追踪用户地理位置的机会，这些地理位置数据可以用来进行数据分析。一种用于编码并交换地理数据结构的常见数据格式是 GeoJSON。正如名字所指，这种格式是基于 JSON 的语法。

绘制交互式地图的一个流行 JavaScript 库是 Leaflet。可以用 Folium 来连接 JavaScript 和 Python。它是一个 Python 库，使用 GeoJSON 格式的数据在 Leaflet.js 地图上对地理数据进行可视化。

值得一提的还有第三方服务（如 Plotly），它们也支持自动生成数据可视化，减轻了创建互动组件的工作量。具体来说，Plotly 还用其 Python 客户端为创建自定义数据可视化提供了支持。Plotly 的图像是在线存放的并且与用户账户相关联（公开存放免费，私有存放收费）。

1.4 Python 中的数据处理

前面介绍了用于数据分析的最重要的 Python 包，现在我们回头来学习一些导入和操作不同格式数据的 Python 工具。

大多数社会媒体 API 都提供 JSON 或 XML 格式的数据。Python 可以使用标准库来处理这些格式。

简便起见，我们先介绍 JSON 这种格式，因为它可以很好地映射到 Python 字典，而且易于阅读和理解。JSON 库的接口也非常简单，你可以将 JSON 数据导入 Python 字典，或者将 Python 字典转储成 JSON。

我们来查看以下代码。

```
# Chap01/demo_json.py
import json

if __name__ == '__main__':
  user_json = '{"user_id": "1", "name": "Marco"}'
  user_data = json.loads(user_json)
  print(user_data['name'])
  # 输出： Marco

  user_data['likes'] = ['Python', 'Data Mining']
  user_json = json.dumps(user_data, indent=4)
  print(user_json)
  """
  输出：
    {
       "user_id": "1",
       "name": "Marco",
       "likes": [
          "Python",
          "Data Mining"
       ]
    }
  """
```

`json.loads()`和`json.dumps()`函数分别将 JSON 字符串转换为 Python 字典以及从 Python 字段转换回 JSON 字符串。另外两个函数是 `json.load()`和 `json.dump()`，它们处理的是文件指针，可以帮助你从文件导出 JSON 数据，或将 JSON 数据存储到文件中。

`json.dumps()`函数还接收第二个参数 `indent` 来指定缩进的字符数量，这对于漂亮的打印效果非常有用。

当手动分析更复杂的 JSON 文件时，使用一个外部 JSON 阅读器很可能更方便，这种阅读器可以在浏览器中良好地打印，并允许用户任意地折叠或展开 JSON 数据。

有些免费的 JSON 阅读工具是基于 Web 的服务，如 JSON Viewer。用户只需粘贴一段 JSON 或者 JSON 数据的 URL，该阅读器就会导入 JSON 数据，并以友好的格式进行展示。

图 1-8 展示了 JSON Viewer 如何显示前面示例中的 JSON 文档。

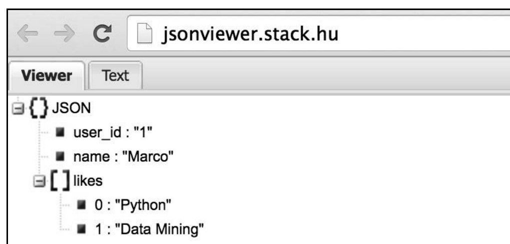


图 1-8　在 JSON Viewer 中漂亮地打印 JSON 数据的示例

如图 1-8 所示，`likes` 字段是一个列表，可以折叠起来以隐藏其元素并简化可视化。上例中的数据很少，但是当文档中包含多个嵌套层时，该功能会非常实用。



当使用基于 Web 的服务或者浏览器扩展时，导入很大的 JSON 文档进行打印会阻塞你的浏览器并使系统变慢。

1.5　创建复杂的数据管道

当创建的数据处理工具大到不仅仅是简单的脚本时，将数据预处理任务分割成较小的单元是非常实用的，我们可以将其映射到数据管道的所有步骤和依赖中。

数据管道是指一系列的数据处理操作，其中包括清洗、填充和操作原始数据，将原始数据转换为分析引擎可以处理的格式。所有的数据分析项目都需要由一系列步骤组成的数据管道。

在原型阶段，比较常见的做法是将这些步骤分割成不同的脚本，然后分别运行，如下所示。

```
$ python download_some_data.py
$ python clean_some_data.py
$ python augment_some_data.py
```

以上每个脚本运行完后和以下脚本产生的输出是一致的，因此不同的步骤之间存在依赖关系。以下脚本集合了所有数据处理步骤，运行如下。

```
$ python do_everything.py
```

以上这个脚本包含的代码可能和以下代码类似。

```
if __name__ == '__main__':
  download_some_data()
  clean_some_data()
  augment_some_data()
```

以上代码中的每个函数都包含了原来单个脚本的逻辑。这种方法的问题是，数据管道中可能会发生错误，因此我们还需要加入很多样板代码，用 try 和 except 来控制可能发生的异常。另外，将这种代码参数化可能会显得有些笨拙。

总之，当从原型发展到更稳定的状态时，需要考虑使用一个数据管理器（也称作工作流管理器）。Python 提供的 Luigi 就是这样一个工具，它是 Spotify 引入的一个开源项目。使用 Luigi 这样的数据管理器的优点如下所示。

- 任务模板：每个数据任务被定义为一个类，其中包含了定义任务如何执行的一些方法、任务的依赖及输出
- 依赖图：一个可视化工具，可以帮助数据工程师可视化并理解数据依赖关系
- 故障恢复：如果数据管道在执行任务的途中发生异常，那么可以从最后的一致状态重新启动
- 与命令行界面以及系统作业调度器（如 cron job）整合
- 个性化异常分析报表

我们不会深入介绍 Luigi 的所有功能，这超出了本书的范围。但是希望你可以了解这个工具，并用它生成更优雅、可重用、易于维护和可扩展的数据管道。

1.6　小结

本章介绍了“如何使用 Python 进行社会媒体的数据挖掘”这个话题的很多方面。我们介绍了其中的一些挑战和机遇，它们使得这个话题研究起来很有趣，并且对于想从社会媒体数据中获得有意义信息的企业来说很有价值。

我们还探讨了社会媒体挖掘的整个流程，其中包括如何用 OAuth 进行鉴权。此外还详细介绍了数据挖掘者的数据工具箱中必备的 Python 工具。根据要分析的社会媒体平台，以及关注的信息类型，Python 提供了各种稳健且成熟的包来实现机器学习、自然语言处理和社会网络分析。

在 1.3.1 节中，我们建议你通过 virtualenv 创建 Python 开发环境，因为这样有助于保持全局开发环境的整洁。

下一章将介绍 Twitter，重点探讨如何用 Twitter API 获取 Twitter 数据，以及如何对 Twitter 数据分段和切分以获得有趣的信息。

第 2 章

Twitter 数据挖掘——标签、话题和时间序列

本章介绍 Twitter 数据的挖掘，包含如下主题：

- ❑ 用 Tweepy 与 Twitter API 进行交互
- ❑ Twitter 数据——推文的结构
- ❑ 分词和频率分析
- ❑ 推文中的话题标签和用户提及
- ❑ 时间序列分析

2.1 入门

Twitter 是最著名的在线社交网络之一，近些年广受欢迎。它提供的服务称为**微博客**，即一种内容非常短的博客变体，每条**推文**和短消息一样最多包含 140 个字符。与其他社会媒体平台（如 Facebook）不同的是，Twitter 网络不是双向的，也就是说用户间的关系不是相互的：你可以关注那些并未关注你的用户，也可以不关注那些关注你的用户。

传统媒体也正在通过社会媒体获取更广泛的受众，并且大多数名人都有一个与粉丝联系的 Twitter 账户。用户可以讨论实时发生的事情，包括庆典、电视节目、体育赛事、政治选举等。

Twitter 还提供了话题标签，以便将对话进行分组，使得用户可以关注特定的话题。话题标签是以`#`符号为前缀的关键词，如`#halloween`（用于分享万圣节服装的图片）或者`#WaterOnMars`（用于标记 NASA 宣布其在火星上找到了水的迹象）。

因为用途广泛，所以 Twitter 对于数据挖掘者来说就是一个潜在的“金矿”，接下来我们将开始挖矿之旅！

2.2　Twitter API

Twitter 提供了一系列的 API，以便我们获取 Twitter 数据，包括读取推文、获取用户资料，以及代表用户发布内容。

为了创建一个获取 Twitter 数据的项目，需要先完成以下两个前期准备步骤：

- 注册我们的应用
- 选择一个 Twitter API 客户端

注册步骤需要花几分钟完成。假设我们已经登录了 Twitter 账户，只需要用浏览器访问应用管理页面并创建一个新的应用。

注册应用后，在 Keys and Access Tokens 选项卡下找到鉴权应用所需的信息。Consumer Key 和 Consumer Secret（也分别称作 API Key 和 API Secret）是你的应用的一个设置。Access Token 和 Access Token Secret 是你的用户账户的一个设置。你的应用可以通过访问令牌请求访问几个用户。这些设置的**访问等级**定义了应用代表用户与 Twitter 交互时可以做哪些事情：只读是一个非常保守的选项，在这种模式下，应用不能发布任何内容，也不可以通过直接消息与其他用户交互。

2.2.1　接口访问频率限制

Twitter API 限制了对应用的访问。这些限制是基于每位用户的，更准确地说，是基于每个访问令牌的。这就意味着，当一个应用使用应用专用的鉴权时，接口访问频率限制是针对整个应用的；而对于基于每位用户的鉴权方法，应用可以提高对 API 的全局访问次数。

熟悉接口访问频率限制的概念非常重要，详见官方文档（https://dev.twitter.com/rest/public/rate-limiting）。还需要考虑不同的 API 会有不同的接口访问频率限制（https://dev.twitter.com/rest/public/rate-limits）。

达到 API 访问次数的上限后，Twitter 会返回一个错误消息，而不是我们请求的数据。如果继续进行更多的 API 访问，再次获得正常接入的时间会变得更长，因为 Twitter 会将我们标记为潜在的恶意用户。当应用需要多次访问 API 时，需要找到一种方法来避免上述情况。在 Python 中，标准库中的 `time` 模块允许我们在执行代码的过程中用 `time.sleep()` 函数设置任意的等待。例如，以下是一段伪代码。

```
# 假设 first_request()和 second_request()是已经定义好的
# 它们将执行 API 请求
import time

first_request()
```

```
time.sleep(10)
second_request()
```

在这个示例中，第一个请求执行 10 秒（`sleep()`参数指定的）后才会执行第二个请求。

2.2.2　搜索与流

Twitter 提供的并不是单个 API。实际上，Twitter 提供了多种方式来获取 Twitter 数据。简便起见，这些 API 可以分为两类：REST API 和流 API（见图 2-1）。

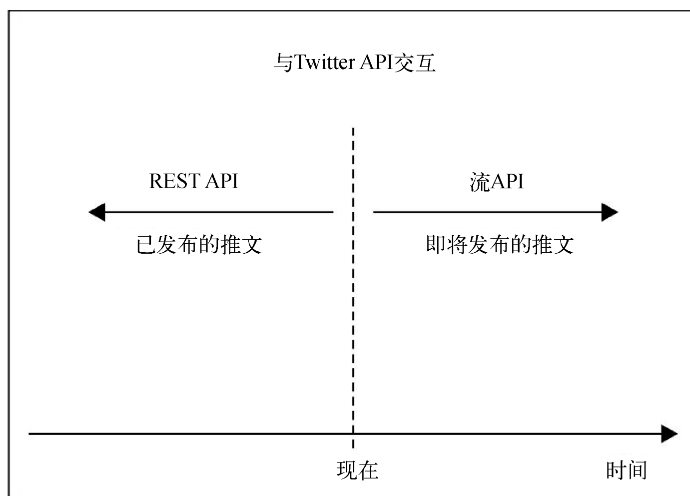


图 2-1　搜索与流的时间维度

如图 2-1 所示，这两种 API 的差别十分简单：所有的 REST API 只允许你回溯时间。当通过一个 REST API 与 Twitter 交互时，我们可以搜索现有的推文，即已经发布的、可以搜索到的推文。通常情况下，这些 API 会限制我们可以检索的推文数量，而且并不是以前面介绍的接口访问频率来限制，而是以时间区间的方式。实际上，通常可以回溯约一周的内容，而更早的内容是无法检索的。需要注意的另一点是，REST API 并不能确保提供 Twitter 上发布的所有推文，因为有些推文不能被检索或者索引上有延迟。

另一方面，流 API 关注的是未来。打开一个连接后，我们可以在接下来的时间内保持其为打开状态。通过保持 HTTP 连接状态打开，我们可以检索符合筛选条件的所有推文。

由于与 Twitter 平台的交互是受限制的，流 API 通常被认为是下载大量推文的更佳方式。采用这种方法的缺点是会消耗大量时间，因为要等推文发布出来才能进行收集。

总的来说，当想要搜索特定用户发布的推文或者获取自己的时间轴时，REST API 非常实用。当希望过滤特定关键词并下载与该关键词（如实时事件）相关的大量推文时，流 API 非常有用。

2.3　从 Twitter 收集数据

为了与 Twitter API 交互，我们需要一个可以对 API 进行不同调用的 Python 客户端。可以从官方文档（https://dev.twitter.com/overview/api/twitter-libraries）中看到好几种方法。这些方法都不是 Twitter 官方维护的，而是由开源社区支持的。虽然很多选项的性能几乎是相当的，但这里我们选择 Tweepy，因为它提供了许多功能，而且目前仍然有积极的维护。

可以通过 pip 安装这个库。

```
$ pip install tweepy==3.3.0
```



Python 3 的兼容性

我们指定安装 3.3 版本的 Tweepy，因为最新版本的 Tweepy 与 Python 3 存在兼容问题，无法在 Python 3.4 的环境中运行示例代码。该问题在撰写本书时还未解决，但是应该会很快解决。

与 Twitter API 交互的第一个部分就是前面介绍过的设置鉴权。注册应用后，你应该拥有消费者密钥、消费者密码、访问令牌和访问令牌密码。

为了分离应用逻辑和配置，我们将用户凭证存储到环境变量中。为什么不直接将这些值硬编码为 Python 代码中的变量呢？十二要素应用（The Twelve-Factor App）的宣言中总结了几个原因。用环境存储配置信息是与语言和操作系统无关的。在不同的部署中改变配置（例如，用个人用户凭证进行局部测试和用公司账户进行生产）并不需要改变代码库本身。更重要的是，环境变量不会意外地带入源代码管理系统，从而避免了对每个人可见。

在 Unix 环境（如 Linux 或 macOS）中，如果你的 shell 是 Bash，那么可以通过以下方式设置环境变量。

```
$ export TWITTER_CONSUMER_KEY="your-consumer-key"
```

在 Windows 环境中，可以通过以下方式在命令行中设置环境变量。

```
$ set TWITTER_CONSUMER_KEY="your-consumer-key"
```

该命令要重复执行 4 次，因为有 4 个变量。

一旦创建好环境，我们将用于创建 Twitter 客户端的 Tweepy 调用封装到两个函数中：一个函数用于读取环境变量和执行鉴权，另一个函数用于创建与 Twitter 交互的 API 对象。

```
# Chap02-03/twitter_client.py
import os
import sys
from tweepy import API
from tweepy import OAuthHandler
```

```
def get_twitter_auth():
  """设置 Twitter 鉴权信息

  返回值：tweepy.OAuthHandler 对象
  """
  try:
    consumer_key = os.environ['TWITTER_CONSUMER_KEY']
    consumer_secret = os.environ['TWITTER_CONSUMER_SECRET']
    access_token = os.environ['TWITTER_ACCESS_TOKEN']
    access_secret = os.environ['TWITTER_ACCESS_SECRET']
  except KeyError:
    sys.stderr.write("TWITTER_* environment variables not set\n")
    sys.exit(1)
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_token, access_secret)
  return auth

def get_twitter_client():
  """配置 Twitter API 客户端

  返回值：tweepy.API 对象
  """
  auth = get_twitter_auth()
  client = API(auth)
  return client
```

get_twitter_auth()函数负责鉴权。try/except 块用于读取环境变量。os 模块包含了一个名为 os.environ 的字典，可以通过一个密钥访问，就像常规字典一样。如果缺失 TWITTER_*环境变量中的某一个，试图访问该密钥会抛出 KeyError 异常，我们将捕捉这个异常以显示错误消息并退出应用。

该函数被 get_twitter_client()调用，它用于创建 tweepy.API 的一个实例，后者可用于多种 Twitter 交互类型。之所以将该逻辑分解为两个单独的函数，是因为鉴权代码也可以在流 API 中重用，接下来将介绍这一点。

2.3.1 从时间线获取推文

客户端鉴权完成后，就可以用客户端下载推文了。

首先思考一个简单场景：如何获取你自己主时间线上的前 10 条推文？

```
from tweepy import Cursor
from twitter_client import get_twitter_client

if __name__ == '__main__':
  client = get_twitter_client()

  for status in Cursor(client.home_timeline).items(10):
    # 处理单个状态
    print(status.text)
```

作为一名 Twitter 用户，你的主时间线就是登录 Twitter 后看到的屏幕。它包含了你关注的一系列账号的推文，最新、最有趣的推文在最上面。

以上代码片段展示了如何使用 `tweepy.Cursor` 对主时间线上的前 10 项进行遍历。首先，需要导入 `Cursor` 和之前定义的 `get_twitter_client` 函数。在主代码块中，我们将用这个函数来创建客户端，该客户端提供了 Twitter API 的访问接口。需要特别说明的是，我们需要用 `home_timeline` 属性来访问自己的主时间线，它将作为参数传入 `Cursor`。

`tweepy.Cursor` 是一个**可迭代的**对象，也就是说，它提供了一个对不同对象执行迭代和翻页的易用接口。前面的示例展示了一个允许开发者对对象本身进行循环的简单抽象，而且开发者无须担心 Twitter API 的请求是如何产生的。



可迭代对象

Python 中的可迭代对象一次可以返回它的一个成员。Python 内置的数据类型（如列表和字典）都是可迭代对象，实现了`__iter__()`或`__getitem__()`方法的其他类的对象也都是可迭代的。

迭代中使用的 `status` 变量表示 `tweepy.Status` 的一个实例，它是 Tweepy 用于封装状态（即推文）的一个模型。在前面的代码片段中，我们只使用了其文本，但是该对象具有很多属性。2.3.2 节中介绍了这些属性。

与其将文本打印到屏幕上，我们更愿意将从 API 检索到的推文存储起来，以便稍后分析。

我们将重构前面的代码片段，以便从自己的主时间线获取推文，然后将 JSON 信息保存到一个文件中。

```
# Chap02-03/twitter_get_home_timeline.py
import json
from tweepy import Cursor
from twitter_client import get_twitter_client

if __name__ == '__main__':
  client = get_twitter_client()

  with open('home_timeline.jsonl', 'w') as f:
    for page in Cursor(client.home_timeline, count=200, include_rts=True).pages(4):
      for status in page:
        # 处理单个状态
        f.write(json.dumps(status._json)+"\n")
```

执行上述代码后会生成一个名为 home_timeline.jsonl 的文件。



JSON Lines 格式

前面的示例中生成的文件带有.jsonl 后缀，而不是.json 后缀。实际上，该文件是 **JSON Lines 格式**的。在这种格式中，文件的每一行都是一个有效的 JSON 文档。如果试图用 `json.loads()` 导入这个文件的全部内容，会抛出 `ValueError` 异常，因为整个内容并不是一个有效的 JSON 文档。如果要使用处理有效 JSON 文档的函数，需要一次处理一行。

JSON Lines 格式特别适合大规模处理：很多大数据框架都允许开发者将输入文件分割成多个片段，从而通过不同的工作进程进行并行处理。

我们在以上代码中迭代了 4 个页面，其中包含 200 条推文，而且每条推文都声明在光标的 `count` 参数中。这样设置的原因是 Twitter 有限制，最多只能从主时间线中检索最新的 800 条推文。

如果从特定用户的时间线中检索推文，即使用 `user_timeline` 方法，而不是 `home_timeline`，那么推文的限制放宽到了 3200 条。

以下脚本可以获取特定用户的时间线。

```
# Chap02-03/twitter_get_user_timeline.py
import sys
import json
from tweepy import Cursor
from twitter_client import get_twitter_client

def usage():
  print("Usage:")
  print("python {} <username>".format(sys.argv[0]))

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  user = sys.argv[1]
  client = get_twitter_client()

  fname = "user_timeline_{}.jsonl".format(user)

  with open(fname, 'w') as f:
    for page in Cursor(client.user_timeline, screen_name=user,
                       count=200).pages(16):
      for status in page:
        f.write(json.dumps(status._json)+"\n")
```

为了执行以上代码，我们需要提供一个命令行参数来指定用户名。例如，如果想要检索我的整个时间线（目前还没有达到 3200 条推文的限制），可以执行以下命令。

```
$ python twitter_get_user_timeline.py marcobonzanini
```

由于我的时间线上内容较少，可以从 Packt 出版社的账号`@PacktPub` 获取更多推文。

```
$ python twitter_get_user_timeline.py PacktPub
```

与前面用主时间线看到的一样，该脚本会生成一个每行带有一个 JSON 文档的.jsonl 文件。如果达到 3200 条的上限，文件的大小约为 10MB。

获取用户时间线的代码和检索主时间线的代码在概念上非常相似。唯一的区别在于前者需要一个命令行参数来指定我们感兴趣的用户。

到目前为止，我们通过 Tweepy 界面使用过的单条推文的唯一属性是`_json`，它用于存储原始的 JSON 响应。接下来将详细介绍推文的结构，然后介绍从 Twitter 获取数据的其他方式。

2.3.2　推文的结构

推文是一个复杂的对象。表 2-1 汇总了推文的所有属性及其含义。一条推文的 API 响应的内容全部包含在可以导入 Python 字典的`_json`属性中。

表 2-1　推文的属性及其含义

属性名称	描　　述
`_json`	JSON 状态响应的字典
`author`	`tweepy.models.User` 实例，表示推文作者
`contributors`	如果启用，该特征是一个贡献者列表
`coordinates`	GeoJSON 格式的坐标字典
`created_at`	`datetime.datetime` 实例，表示推文的生成时间
`entities`	推文中 URL、话题标签和提及信息的字典
`favorite_count`	推文被标记为喜欢的次数
`favorited`	表示鉴权用户是否喜欢该条推文
`geo`	坐标（不建议使用，可以用 `coordinates` 替代）
`id`	推文的唯一 ID，是一个较大的整数
`id_str`	推文的唯一 ID，是一个字符串
`in_reply_to_screen_name`	推文正在回复的状态的用户名
`gein_reply_to_status_id`	推文正在回复的状态的 ID，是一个较大的整数
`in_reply_to_status_id_str`	推文正在回复的状态的 ID，是一个字符串
`in_reply_to_user_id`	推文正在回复的状态的用户 ID，是一个较大的整数
`in_reply_to_user_id_str`	推文正在回复的状态的用户 ID，是一个字符串
`is_quote_status`	表示推文是否为引用（即包含另一条推文）
`lang`	带有推文的语言代码的字符串
`place`	`tweepy.models.Place` 实例，表示推文中附加的地点
`possibly_sensitive`	表示推文是否包含可能带有敏感内容的 URL
`retweet_count`	状态被转发的次数

（续）

属性名称	描 述
retweeted	表示状态是否为转发推文
source	描述用于发布状态的工具的字符串
text	带有状态内容的字符串
truncated	表示状态是否被截短（例如，转发超过 140 个字符）
user	tweepy.models.User 实例，表示推文的作者（不建议使用，可以用 author 替代）

带有 ID（如用户 ID 或推文 ID）的属性会有一个对应的字符串版本。这是很有必要的，因为有些编程语言（如 JavaScript）不支持超过 53 比特的数字，而 Twitter 使用的是 64 比特的数字。为了避免这个问题，Twitter 建议使用*_str 属性。

我们可以看到，并不是所有的属性都可以转换为 Python 的内置类型，如字符串或布尔值。实际上，有些复杂对象（如用户资料）是完全包含在 API 响应中的，而 Tweepy 会将这些对象转换为合适的模型。

以下示例展示了一条由 Packt 出版社发布的示例推文。该推文是原始的 JSON 格式，可通过 API 或者_json 属性获得。（为了简洁起见，省略了一些字段。）

首先，期望格式的生成时间如下所示。

```
{
  "created_at": "Fri Oct 30 05:26:05 +0000 2015",
```

entities 属性是一个字典，其中包含带有标签的实体的不同列表。例如，hashtags 元素展示了给定推文中出现了#Python 话题标签。同样，我们还有照片、URL 以及用户提及信息。

```
"entities": {
  "hashtags": [
    {
      "indices": [
        72,
        79
      ],
      "text": "Python"
    }
  ],
  "media": [
    {
      "display_url": "pic.twitter.com/muZlMreJNk",
      "id": 659964467430735872,
      "id_str": "659964467430735872",
      "indices": [
        80,
        103
      ],
      "type": "photo",
```

```
        "url": "https://t.co/muZlMreJNk"
      }
    ],
    "symbols": [],
    "urls": [
      {
        "indices": [
          48,
          71
        ],
        "url": "https://t.co/NaBNan3iVt"
      }
    ],
    "user_mentions": [
      {
        "id": 80589255,
        "id_str": "80589255",
        "indices": [
          33,
          47
        ],
        "name": "Open Source Way",
        "screen_name": "opensourceway"
      }
    ]
  },
```

以下属性已经在表 2-1 中详细解释过了。我们注意到这条推文缺失了地理信息。我们也可以将前面与实体相关的信息与存储在 `text` 属性中的实际推文进行比较。

```
  "favorite_count": 4,
  "favorited": false,
  "geo": null,
  "id": 659964467539779584,
  "id_str": "659964467539779584",
  "lang": "en",
  "retweet_count": 1,
  "retweeted": false,
  "text": "Top 3 open source Python IDEs by @opensourceway
    https://t.co/NaBNan3iVt #Python https://t.co/muZlMreJNk",
```

`user` 属性是一个字典，表示发送推文的用户，在这个示例中是`@PacktPub`。正如前面提到的，这是一个复杂对象，其中含有推文中嵌入的所有用户相关信息。

```
    "user": {
      "created_at": "Mon Dec 01 13:16:47 +0000 2008",
      "description": "Providing books, eBooks, video tutorials, and
        articles for IT developers, administrators, and users.",
      "entities": {
        "description": {
          "urls": []
        },
        "url": {
```

```
          "urls": [
            {
              "display_url": "PacktPub.com",
              "expanded_url": "http://www.PacktPub.com",
              "indices": [
                0,
                22
              ],
              "url": "http://t.co/vEPCgOu235"
            }
          ]
        }
      },
      "favourites_count": 548,
      "followers_count": 10090,
      "following": true,
      "friends_count": 3578,
      "id": 17778401,
      "id_str": "17778401",
      "lang": "en",
      "location": "Birmingham, UK",
      "name": "Packt Publishing",
      "screen_name": "PacktPub",
      "statuses_count": 10561,
      "time_zone": "London",
      "url": "http://t.co/vEPCgOu235",
      "utc_offset": 0,
      "verified": false
    }
  }
```


这个示例表明分析推文时需要考虑以下两个方面：

- 实体本身是带有标签的
- 用户资料是完全嵌入的

第一点意味着实体分析简化了，即我们不需要显式地搜索主题标签、用户提及、嵌入的 URL、多媒体等实体，因为 Twitter API 已经提供了这些信息，并且一并提供了这些信息在文本中的偏置值（该属性称作 `indices`）。

第二点意味着我们不需要在其他地方存储用户资料信息，并通过外键连接/合并数据。实际上，每条推文中的用户信息是冗余重复的。



处理非标准化数据

嵌入冗余数据的方法与非标准化的概念相关。标准化是关系型数据库设计中的最佳实践，而非标准化用于大规模处理以及属于广泛的 NoSQL 家族的数据库。这种方法背后的基本原理是，冗余存储用户信息只需要微不足道的花费，而避免连接/合并操作带来的收益（性能提升）是持久的。

2.3.3 使用流 API

流 API 是获取大量数据而不超过接口访问频率限制的常用方法之一。我们已经探讨过这种 API 与其他 REST API（特别是 Search API）的差别，也解释过可能需要重新思考我们的应用如何与用户交互。

在深入学习流 API 前，有必要查看流 API 的文档（https://dev.twitter.com/streaming/overview）来理解其特性。

本节，我们将通过扩展 Tweepy 提供的默认 `StreamListener` 类实现一个自定义的流监听器。

```
# Chap02-03/twitter_streaming.py
import sys
import string
import time
from tweepy import Stream
from tweepy.streaming import StreamListener
from twitter_client import get_twitter_auth

class CustomListener(StreamListener):
  """自定义 StreamListener 类获取 Twitter 流数据"""

  def __init__(self, fname):
    safe_fname = format_filename(fname)
    self.outfile = "stream_%s.jsonl" % safe_fname

  def on_data(self, data):
    try:
      with open(self.outfile, 'a') as f:
        f.write(data)
        return True
    except BaseException as e:
      sys.stderr.write("Error on_data: {}\n".format(e))
      time.sleep(5)
    return True

  def on_error(self, status):
    if status == 420:
      sys.stderr.write("Rate limit exceeded\n".format(status))
      return False
    else:
      sys.stderr.write("Error {}\n".format(status))
      return True

def format_filename(fname):
  """将 fname 转换成文件名允许的字符串

  返回值：字符串
  """
  return ''.join(convert_valid(one_char) for one_char in fname)
```

```
def convert_valid(one_char):
  """如果字符为无效值则转换为下划线'_'

  返回值：字符串
  """
  valid_chars = "-_.%s%s" % (string.ascii_letters, string.digits)
  if one_char in valid_chars:
    return one_char
  else:
    return '_'

if __name__ == '__main__':
  query = sys.argv[1:] # 命令行界面参数列表
  query_fname = ' '.join(query) # 字符串
  auth = get_twitter_auth()
  twitter_stream = Stream(auth, CustomListener(query_fname))
  twitter_stream.filter(track=query, async=True)
```


流逻辑的核心在 CustomListener 类中实现，该类继承了 StreamListener，并重写了方法 on_data()和 on_error()。当使用 API 获取数据的过程中碰到错误时，就会触发这两个处理器。

这两个函数的返回类型都是布尔值：True 是继续，False 是停止运行。因此，只在发生严重错误时才使用 False，以便应用可以继续下载数据。这样可以避免微小的错误导致应用终止运行，比如应用端的网络短暂暂停，或者 Twitter 的一个 HTTP 503 错误，后者意味着服务暂时不可用（但会很快恢复）。

on_error()方法会处理来自 Twitter 的显式错误。可以在文档（https://dev.twitter.com/overview/api/response-codes）中查看 Twitter API 的状态代码的完整列表。on_error()方法的实现只在发生 420 错误时终止运行，即达到 Twitter API 接口访问频率限制时终止应用。超过接口访问频率限制越多，能够再次使用该服务之前需要等待的时间就越久。因此，最好停止下载并追查产生问题的原因。其他错误会在 stderr 接口打印。这种打印方式比只使用 print()更好，因为我们可以重定向错误并将其输出到特定文件（如果需要的话）。一种更好的方法是用 logging 模块创建一个恰当的日志处理机制，但这超出了本章的范围，不在此深入介绍。

当成功获取数据时，调用 on_data()方法。该函数将数据存储在一个.jsonl 文件中。这个文件的每一行都将包含一条 JSON 格式的推文。一旦写入数据，我们将返回 True 来继续运行。如果在这个过程中发生任何异常，我们将捕获异常，在 stderr 打印一条消息，并让应用休眠 5 秒，然后再次返回 True 来继续运行程序。异常后的短暂休眠仅仅是为了避免偶尔的网络中断引起应用阻塞。

CustomListener 类用一个辅助函数来清洗查询，并将其作为文件名。format_filename()函数会循环给定字符串的每个字符，并用 convert_valid()函数将非法字符转换为下划线。合

法的字符有：横杠、下划线和点（-、_和.），以及 ASCII 字母和数字。

当运行 `twitter_streaming.py` 脚本时，必须在命令行输入参数。监听器会将以空白分隔的这些参数作为下载推文的关键词。

为了提供一个示例，我运行脚本获取了 2015 年橄榄球世界杯中新西兰和澳大利亚之间决赛的相关推文。在 Twitter 上关注这场球赛的粉丝大多使用了`#RWC2015`话题标签（赛事全程使用的话题标签）和`#RWCFinal`话题标签（用于关注赛事的决赛日）。我将这两个话题标签以及 `rugby` 一词作为流监听器的搜索词。

```
$ python twitter_streaming.py \#RWC2015 \#RWCFinal rugby
```

`#`前面的反斜杠用于转义，因为 shell 用`#`表示注释的开头。转义该字符可以确保该字符串正确地传递给脚本。

赛事设定在**格林威治标准时间**（Greenwich Mean Time，GMT）的 2015 年 10 月 31 日下午 4 点。在下午 3~6 点运行以上脚本产生了近 800MB 的数据，共计 200 000 多条推文。在 Twitter 上的相关讨论在赛事结束后还持续了一段时间，但我们在这 3 小时内搜集的数据已经足够做一些有趣的分析了。

2.4　分析推文——实体分析

本节将分析推文中的实体。我们将对前面搜集到的数据做一些频率分析。数据的切片和分段操作允许用户生成一些有趣的统计，这些统计可以帮助用户获得一些有关数据的有用信息并回答一些问题。

分析话题标签这样的实体非常有趣，因为这些注解有助于用户以一种明确的方式标记推文的话题。

接下来分析 Packt 出版社的推文。因为 Packt 出版社支持并推动开源软件，所以我们希望找到 Packt 出版社经常提及哪些技术。

以下脚本从一个用户时间线抽取话题标签，并生成了一个最常用的话题标签列表。

```
# Chap02-03/twitter_hashtag_frequency.py
import sys
from collections import Counter
import json

def get_hashtags(tweet):
  entities = tweet.get('entities', {})
  hashtags = entities.get('hashtags', [])
  return [tag['text'].lower() for tag in hashtags]
```

```
if __name__ == '__main__':
  fname = sys.argv[1]
  with open(fname, 'r') as f:
    hashtags = Counter()
    for line in f:
      tweet = json.loads(line)
      hashtags_in_tweet = get_hashtags(tweet)
      hashtags.update(hashtags_in_tweet)
    for tag, count in hashtags.most_common(20):
      print("{}: {}".format(tag, count))
```


可以用以下命令运行上述代码。

```
$ python twitter_hashtag_frequency.py user_timeline_PacktPub.jsonl
```

这里的 user_timeline_PacktPub.jsonl 是我们在前面搜集到的 JSON Lines 文件。

该脚本从命令行接收一个.jsonl 文件名作为参数，并且每次读入一行内容。每一行包含的是一个 JSON 文档，读入后会将这个文档赋给 tweet 变量，并用 get_hashtags()辅助函数抽取话题标签列表。这些实体的类型存储在 collections.Counter 声明过的 hashtags 变量中。collections.Counter 是一个对话题标签对象（这里是字符串）计数的特殊字典。该计数器将字符串作为字典的键，将频率作为值。

作为 dict 的子类，Counter 对象本身也是一个无序集合。most_common()方法负责按照频率大小（频率最大的排第一个）对键进行排序，并返回一个(key, value)二元组列表。

get_hashtags()辅助函数负责从推文中检索出话题标签列表。载入字典数据结构中的整条推文是该函数的唯一参数。如果该推文中存在实体，那么字典中就会包含 entities 键。因为这是一个可选项，所以我们不能直接获取 tweet['entities']，否则会出现 KeyError 异常。因此，我们用 get()函数来获取实体，不存在实体时就返回一个空字典。第二步是从实体中获取话题标签。由于 entities 也是个字典，hashtags 键也是个可选项，我们仍然使用 get()函数，但如果不存在话题标签，就指定返回一个空列表。最后，我们用一个列表推导式来迭代话题标签，以抽取出其中的文本。用 lower()函数归一化话题标签，以便将所有文本转换成小写。这样一来，#Python 和#PYTHON 都将被看作#python。

执行以上代码分析 PacktPub 的推文会生成以下输出。

```
packt5dollar: 217
python: 138
skillup: 132
freelearning: 107
gamedev: 99
webdev: 96
angularjs: 83
bigdata: 73
javascript: 69
unity: 65
```

```
hadoop: 46
raspberrypi: 43
js: 37
pythonweek: 36
levelup: 35
r: 29
html5: 28
arduino: 27
node: 27
nationalcodingweek: 26
```

可以看到，这些都是关于 PacktPub 的事件或促销（如`#packt5dollar`），但大多数话题标签都是关于特定技术的，其中 Python 和 JavaScript 是推文中提及最多的技术。

前面的脚本给出了 PacktPub 最常用的话题标签概览，但我们希望可以更深入一些。实际上，我们能生成一些更具描述性的统计来帮助理解 Packt 出版社是如何使用话题标签的。

```
# Chap02-03/twitter_hashtag_stats.py
import sys
from collections import defaultdict
import json

def get_hashtags(tweet):
  entities = tweet.get('entities', {})
  hashtags = entities.get('hashtags', [])
  return [tag['text'].lower() for tag in hashtags]

def usage():
  print("Usage:")
  print("python {} <filename.jsonl>".format(sys.argv[0]))

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  fname = sys.argv[1]
  with open(fname, 'r') as f:
    hashtag_count = defaultdict(int)
    for line in f:
      tweet = json.loads(line)
      hashtags_in_tweet = get_hashtags(tweet)
      n_of_hashtags = len(hashtags_in_tweet)
      hashtag_count[n_of_hashtags] += 1

    tweets_with_hashtags = sum([count for n_of_tags, count in
                               hashtag_count.items() if n_of_tags > 0])
    tweets_no_hashtags = hashtag_count[0]
    tweets_total = tweets_no_hashtags + tweets_with_hashtags
    tweets_with_hashtags_percent = "%.2f" % (tweets_with_hashtags
                                             / tweets_total * 100)
    tweets_no_hashtags_percent = "%.2f" % (tweets_no_hashtags /
                                           tweets_total * 100)
    print("{} tweets without hashtags
          ({}%)".format(tweets_no_hashtags,
```

```
        tweets_no_hashtags_percent))
  print("{} tweets with at least one hashtag
        ({}%)".format(tweets_with_hashtags,
        tweets_with_hashtags_percent))

  for tag_count, tweet_count in hashtag_count.items():
    if tag_count > 0:
      percent_total = "%.2f" % (tweet_count / tweets_total * 100)
      percent_elite = "%.2f" % (tweet_count / tweets_with_hashtags * 100)
      print("{} tweets with {} hashtags ({}% total, {}%
            elite)".format(tweet_count, tag_count,
            percent_total, percent_elite))
```

可以用以下命令执行以上脚本。

```
$ python twitter_hashtag_stats.py user_timeline_PacktPub.jsonl
```

基于我们搜集的数据，以上命令会产生以下输出结果。

```
1373 tweets without hashtags (42.91%)
1827 tweets with at least one hashtag (57.09%)
1029 tweets with 1 hashtags (32.16% total, 56.32% elite)
585 tweets with 2 hashtags (18.28% total, 32.02% elite)
181 tweets with 3 hashtags (5.66% total, 9.91% elite)
29 tweets with 4 hashtags (0.91% total, 1.59% elite)
2 tweets with 5 hashtags (0.06% total, 0.11% elite)
1 tweets with 7 hashtags (0.03% total, 0.05% elite)
```

可以看到，PacktPub 的大多数推文都至少有一个话题标签，这也验证了这种实体在人们用 Twitter 交流中的重要性。

另外，话题标签个数的分布说明每条推文使用的话题标题的个数不会很多。只有约 1%的推文会使用 4 个及以上的话题标签。在以上统计中可以看到两种百分数：第一个是针对所有推文统计的，第二个是对至少有一条引用的推文（称作精英集合）的统计。

同样，可以用以下代码观察用户的提及情况。

```
# Chap02-03/twitter_mention_frequency.py
import sys
from collections import Counter
import json

def get_mentions(tweet):
  entities = tweet.get('entities', {})
  hashtags = entities.get('user_mentions', [])
  return [tag['screen_name'] for tag in hashtags]

if __name__ == '__main__':
  fname = sys.argv[1]
  with open(fname, 'r') as f:
    users = Counter()
    for line in f:
      tweet = json.loads(line)
```

```
        mentions_in_tweet = get_mentions(tweet)
        users.update(mentions_in_tweet)
    for user, count in users.most_common(20):
        print("{}: {}".format(user, count))
```

可以用以下命令运行上述脚本。

```
$ python twitter_mention_frequency.py user_timeline_PacktPub.jsonl
```

输出结果如下所示。

```
PacktPub: 145
De_Mote: 15
antoniogarcia78: 11
dptech23: 10
platinumshore: 10
packtauthors: 9
neo4j: 9
Raspberry_Pi: 9
LucasUnplugged: 9
ccaraus: 8
gregturn: 8
ayayalar: 7
rvprasadTweet: 7
triqui: 7
rukku: 7
gileadslostson: 7
gringer_t: 7
kript: 7
zaherg: 6
otisg: 6
```

2.5　分析推文——文本分析

前一节中分析了推文的实体字段。这提供了有关推文的一些有用信息，因为这些实体是由推文的作者明确标记的。本节主要关注非结构化数据，即推文的原始文本。我们将介绍文本分析的几个方面，如文本预处理和文本归一化，还将对推文做一些数值分析。在深入学习之前，先介绍一些术语。

分词是预处理阶段最重要的概念之一。给定一个文本流（如一条推文状态），分词是将这个文本分解为各个独立单元（称为词项）的过程。在最简单的形式中，该单元是单词。但我们也可以将文本分解为更复杂的单元，如词组、符号等。

分词听起来是一个琐碎的任务，但自然语言处理社区对其进行了广泛研究。第 1 章对该领域做了简短介绍，并提到了 Twitter 如何改变了分词的规则，因为 Twitter 的内容包含表情符号、话题标签、用户提及和 URL 等内容，这与标准英语非常不同。因此，在使用 NLTK 库时，我们展示了对 Twitter 内容进行分词的工具 `TweetTokenizer` 类。本章也会使用该工具。

需要考虑的另一个预处理步骤是**停用词移除**。停用词是指单独存在时没有任何含义的单词。这类单词包括冠词、介词、副词，等等。频率分析将展示出，这些单词在所有数据集中都是最常见的。虽然可以通过分析数据（如选择在 95%以上的文档中都出现过的词）自动编译一个停用词列表，但更保险的方式通常是只选用常用的英语停用词。NLTK 库提供了一个可以通过 `nltk.corpus.stopwords` 模块获得的常用英语停用词列表。

停用词移除也可以扩展到符号（如标点符号）或专业领域的单词。在我们的 Twitter 内容中，常见的停用词是 RT（retweet 的缩写）和 via（通常用于提到分享内容的作者）。

最后，另一个重要的预处理步骤是归一化。这是一个涵盖性术语，包含多种处理。总的来说，当需要将不同的项聚合到同一单元时，可以使用归一化。大小写归一化是归一化的特例，意味着所有的项都是小写，这样可以将原来具有不同大小写形式的相同字符串内容匹配到一起（如 `'python' == 'Python'.lower()`）。执行大小写归一化的优点是，同一项的频率可以自动聚合到一起，而不是以同一项的不同形式分散开。

可以用以下脚本生成我们项的第一次频率分析。

```
# Chap02-03/twitter_term_frequency.py
import sys
import string
import json
from collections import Counter
from nltk.tokenize import TweetTokenizer
from nltk.corpus import stopwords

def process(text, tokenizer=TweetTokenizer(), stopwords=[]):
  """处理推文文本：
  - 转小写
  - 分词
  - 移除停用词
  - 移除数字

  返回值：字符串数组
  """
  text = text.lower()
  tokens = tokenizer.tokenize(text)
  # 如果需要还原缩略形式，请去掉下面代码的注释
  # tokens = normalize_contractions(tokens)
  return [tok for tok in tokens if tok not in stopwords and not tok.isdigit()]

def normalize_contractions(tokens):
  """英语缩略语还原示例

  返回值：生成器
  """
  token_map = {
    "i'm": "i am",
    "you're": "you are",
```

```
        "it's": "it is",
        "we're": "we are",
        "we'll": "we will",
      }
      for tok in tokens:
        if tok in token_map.keys():
          for item in token_map[tok].split():
            yield item
        else:
          yield tok

if __name__ == '__main__':
  tweet_tokenizer = TweetTokenizer()
  punct = list(string.punctuation)
  stopword_list = stopwords.words('english') + punct + ['rt', 'via']

  fname = sys.argv[1]
  tf = Counter()
  with open(fname, 'r') as f:
    for line in f:
      tweet = json.loads(line)
      tokens = process(text=tweet.get('text', ''),
                       tokenizer=tweet_tokenizer,
                       stopwords=stopword_list)
      tf.update(tokens)
    for tag, count in tf.most_common(30):
      print("{}: {}".format(tag, count))
```

预处理逻辑的核心在 `process()`函数中实现。该函数将一个字符串作为输入，并返回一个字符串列表作为输出。这里只用几行代码就实现了前面提到的所有预处理步骤，包括大小写归一化、分词、停用词移除。

该函数还有两个可选参数：一个分词器和一个停用词列表。前者是一个实现 `tokenize()`方法的对象，后者允许自定义停用词移除的过程。当使用停用词移除时，该函数也可以通过对字符串使用 `isdigit()`函数去除数字（如`'5'`或`'42'`）。

以上脚本需要从命令行输入.jsonl 文件名作为参数。它初始化 `TweetTokenizer` 来进行分词，然后定义一个停用词列表。这个列表是由 NLTK 提供的常见英语停用词以及 `string.punctuation` 中定义的标点符号组成的。为了完善停用词列表，我们还包括了`'rt'`、`'via'`和`'...'`词项。（最后的`'...'`是一个单字符的 Unicode 符号，表示水平省略号。）

可以用以下命令运行该脚本。

```
$ python twitter_term_frequency.py filename.jsonl
```

输出结果如下所示。

```
free: 477
get: 450
```

```
today: 437
ebook: 342
http://t.co/wrmxoton95: 295
-: 265
save: 259
ebooks: 236
us: 222
http://t.co/ocxvjqbbiw: 214
#packt5dollar: 211
new: 208
data: 199
@packtpub: 194
': 192
hi: 179
titles: 177
we'll: 160
find: 160
guides: 154
videos: 154
sorry: 150
books: 149
thanks: 148
know: 143
book: 141
we're: 140
#python: 137
grab: 136
#skillup: 133
```

正如我们所见,输出结果中包含单词、话题标签、用户提及、URL 和用 `string.punctuation` 未获取到的 Unicode 符号。对于这些额外的符号,我们可以扩展停用词列表来获取这些标点符号。`TweetTokenizer` 可以帮助我们获取话题标签和用户提及，因为这些项词都是有效的。但是其中 `we're` 和 `we'll` 并不是我们期望的词项，因为它们是两个独立词项的缩略形式，而不是一个独立的词项。如果将缩略形式展开（如 `we are` 和 `we will`），得到的就是一个停用词序列，因为这些缩略形式通常是代词和常见动词。英文缩略形式的完整列表可以参见维基百科（Wikipedia: List of English contractions）。

处理这些英语缩略形式的一种方法是将其归一化为扩展形式。例如，以下函数接收一个词项列表作为输入，并返回一个归一化后的词项列表（更准确地说是一个生成器，因为我们使用了 `yield` 关键字）。

```
def normalize_contractions(tokens):
  token_map = {
    "i'm": "i am",
    "you're": "you are",
    "it's": "it is",
    "we're": "we are",
    "we'll": "we will",
  }
```

```
  for tok in tokens:
    if tok in token_map.keys():
      for item in token_map[tok].split():
        yield item
    else:
      yield tok
```

以下示例展示了如何在交互式的解释器中运行上述程序。

```
>>> tokens = tokens = ["we're", "trying", "something"]
>>> list(normalize_contractions(tokens))
['we', 'are', 'trying', 'something']
```

这种方法的主要问题是，要手动指定需要处理的缩略形式。虽然这些缩略形式的数量有限，但将一切都转化成字典将是一项繁琐的工作。更重要的是，还需要做一些消除歧义的工作，例如，我们前面碰到的 we'll 这种缩略形式可以映射为 we will 或 we shall，当然，这个列表可能会更长。

另一个方法是将这些词项作为停用词，因为在归一化后，组成这些缩略词的成分基本上都是停用词。

最佳的实践方式可能取决于你的应用，因此关键的问题是，process()函数做完预处理后会发生什么。对于现阶段来说，也可以在预处理后不做任何改变，即不处理这些缩略词。



yield 和生成器

在上述示例中，normalize_contractions()函数使用了 yield 关键字，而不是 return 关键字。yield 关键字用于产生一个**生成器**，它是一个只能够迭代一次的迭代器，因为它不会在内存中存储任何项，而是在运行过程中生成这些项。这种处理方式的优点之一就是减少了内存消耗，因此，我们建议对大对象的迭代采用这种方法。该关键字还允许在同一个函数调用中生成多个项，调用关键字 return 后会结束运算（例如，normalize_contractions()中的 for 循环只会针对第一个词项运行）。

总的来说，本节从一个独特的角度分析了项和实体的频率。

我们在 twitter_hashtag_frequency.py 和 twitter_term_frequency.py 中给出了源代码，可以修改这些脚本，将打印最高频的项改为画出分布图像。

```
# Chap02-03/twitter_term_frequency_graph.py
y = [count for tag, count in tf.most_common(30)]
x = range(1, len(y)+1)

plt.bar(x, y)
plt.title("Term Frequencies")
plt.ylabel("Frequency")
plt.savefig('term_distribution.png')
```

上述代码段展示了如何画出 `twitter_term_frequency.py` 中各项的频率分布，同样也很容易将其用于话题标签的频率分布。对`@PacktPub` 的推文运行上述脚本后，可以得到一个 term_distribution.png 文件，如图 2-2 所示。

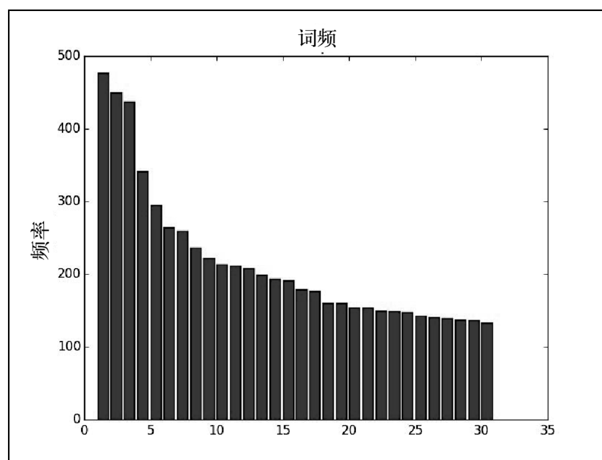


图 2-2　`@PacktPub` 近期推文中各项的频率分布

图 2-2 并没有显示出每项的名称，因为以上分析主要关注的是频率分布。正如我们在图中看到的，左边一些项的频率很高，最高频的项至少是第 10 位后各项频率的 2 倍。当向图的右方移动时，曲线变得不那么陡峭，这意味着右边的项有着相似的频率。

稍做修改，如果用 `most_common(1000)`（即最高频的 1000 个项）运行同样的代码，可以更清晰地看到这种现象，如图 2-3 所示。

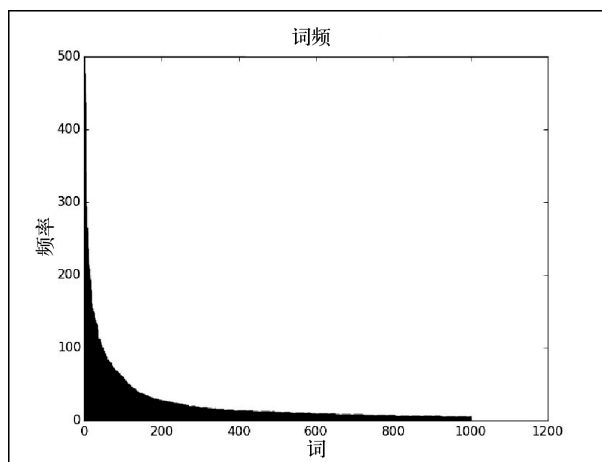


图 2-3　`@PacktPub` 的推文中最高频的 1000 个项的频率分布

我们在图 2-3 中看到的曲线近似体现了**幂法则**。在统计学中，幂法则是两个量之间的一种函数关系，这里指的是项的频率及其在频率排名中的位置。这种类型的分布总是呈现一个**长尾**，意思是一小部分的项主导了整体分布，而整个分布中还有很多低频项。这种现象的另一个名称是**二八定律**或**帕累托法则**，指的是约 80%的实际效果是由 20%的原因产生的。（在以上示例中，20%的项决定了 80%的频率。）

几十年前，美国语言学家 George Zipf 推广了如今所谓的 **Zipf 定律**。这条经验定律表明，给定一个文档集合，任何单词的频率与其在频率表中的排名成反比。换句话说，最高频率单词的出现次数将是第二高频率单词的 2 倍，是第三高频率单词的 3 倍，如此类推。实际上，这条定律描述了一个趋势，而不是精确的频率。有趣的是，Zipf 定律适用于很多自然语言以及社会科学中与语言相关的很多排序研究。

2.6　分析推文——时间序列分析

上一节分析了推文的内容。本节将介绍 Twitter 数据分析的另一个有趣的方面——推文随时间的分布。

通常来说，时间序列是数据点的一个序列，该序列包含给定时间段内的连续观察值。由于 Twitter 提供了一个包含推文的精确时间戳的 `created_at` 字段，我们可以在一个临时的存储桶中重排这些推文，以观察用户对实时事件的反应。我们希望观察一群用户（而不是一个用户）是如何发推文的，因此，通过流 API 收集的数据是最适合做这种分析的。

本节的分析使用了 2015 年橄榄球世界杯决赛的数据集。这个示例很好地展示了用户是如何对实时事件（如体育赛事、音乐会、政治选举、重大灾难、电视节目等）做出反应的。对 Twitter 进行时间序列分析的另一个应用是网络声誉管理。公司可能想监测用户（可能是顾客）在社会媒体上对其做出的评价，例如追踪用户对新发布产品的反应，而 Twitter 的动态特性使其成为了极佳的追踪工具。

我们将用 pandas 的功能来操作时间序列，并用 matplotlib 完成序列的可视化。

```
# Chap02-03/twitter_time_series.py
import sys
import json
from datetime import datetime
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

import pandas as pd
import numpy as np
import pickle

if __name__ == '__main__':
  fname = sys.argv[1]
```

```
with open(fname, 'r') as f:
  all_dates = []
  for line in f:
    tweet = json.loads(line)
    all_dates.append(tweet.get('created_at'))
  ones = np.ones(len(all_dates))
  idx = pd.DatetimeIndex(all_dates)
  # 实际序列值（开始都用 1 表示）
  my_series = pd.Series(ones, index=idx)

  # 取样/分桶 1 分钟数据
  per_minute = my_series.resample('1Min', how='sum').fillna(0)
  # 绘制序列值
  print(my_series.head())
  print(per_minute.head())

  fig, ax = plt.subplots()
  ax.grid(True)
  ax.set_title("Tweet Frequencies")

  hours = mdates.MinuteLocator(interval=20)
  date_formatter = mdates.DateFormatter('%H:%M')

  datemin = datetime(2015, 10, 31, 15, 0)
  datemax = datetime(2015, 10, 31, 18, 0)

  ax.xaxis.set_major_locator(hours)
  ax.xaxis.set_major_formatter(date_formatter)
  ax.set_xlim(datemin, datemax)
  max_freq = per_minute.max()
  ax.set_ylim(0, max_freq)
  ax.plot(per_minute.index, per_minute)

  plt.savefig('tweet_time_series.png')
```

可以用以下命令运行以上代码。

```
$ python twitter_time_series.py stream__RWC2015__RWCFinal_Rugby.jsonl
```

该脚本读取作为命令行参数的.jsonl 文件，一次导入一行推文。因为只对推文的发布时间感兴趣，所以我们创建一个列表 `all_dates`，其中包含每条推文的 `created_at` 属性。

pandas 序列的索引是创建时间，在 `idx` 变量中表示为前面搜集的日期 `pd.DatetimeIndex`。这些日期的颗粒度可以到秒。然后我们用 `ones()`函数创建了由 1 组成的 `np.array`，以聚合推文频率。

创建好序列后，用名为分桶抽样或重抽样的方法改变我们索引序列的方式，这里以分钟为单位对推文进行分组。因为指定了通过 `sum` 进行重抽样，所以每个桶将包含一分钟内发布的推文数目。在重抽样的末尾加上 `fillna(0)`以防止桶中没有任何推文。这里，桶不会被丢弃，而是用 0 表示其频率。当然，我们这个大小的数据集不会发生这种情况，但在小数据集上是有可能发生的。

为了明确重抽样后发生了什么,我们可以观察通过打印 my_series.head() 和 per_minute.head() 得到的以下示例。

```
# 重抽样之前
# my_series.head()
2015-10-31 15:11:32 1
2015-10-31 15:11:32 1
2015-10-31 15:11:32 1
2015-10-31 15:11:32 1
2015-10-31 15:11:32 1
dtype: float64
# 重抽样之后
# per_minute.head()
2015-10-31 15:11:00 195
2015-10-31 15:12:00 476
2015-10-31 15:13:00 402
2015-10-31 15:14:00 355
2015-10-31 15:15:00 466
dtype: float64
```

输出的第一列包含序列索引，它是一个 datetime 对象（时间精确到秒），而第二列是与该索引相关的值（频率）。

由于每秒的推文数量很大，原始的序列 my_series 多次展示了同样的索引（如 15:11:32）。一旦该序列被重抽样，序列索引的颗粒度就会降到以分钟为单位。

随后的代码用 matplotlib 库创建了时间序列分析的良好可视化。总的来说，matplotlib 比其他数据分析库要繁琐，但并不复杂。在这个示例中，有趣的部分是用 MinuteLocator 和 DateFormatter 来正确标记 x 轴的时间间隔，这里以 20 分钟为时间间隔。

图像被保存在 tweet_time_series.png 中，如图 2-4 所示。

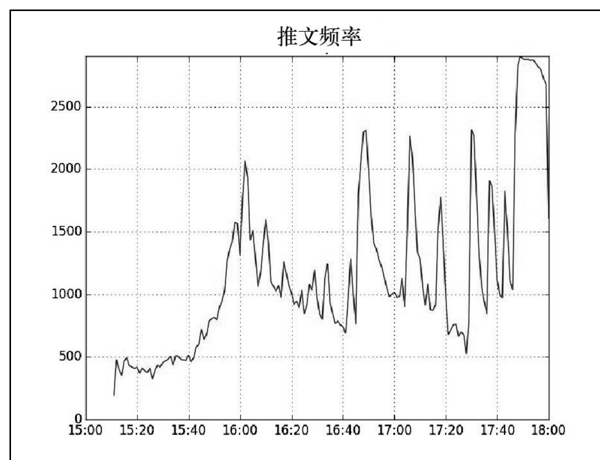


图 2-4　2015 年橄榄球世界杯决赛期间的推文频率分布

图 2-4 中的时区是英国格林威治标准时间。决赛时间是下午 4 点。正如我们看到的，当临近开赛时间时，用户活跃度增强，并且开赛后不久就达到了峰值。在几个话题性时刻，峰值超过了每分钟 2000 条推文，如新西兰大比分领先以及澳大利亚队急起直追时（对于中立的粉丝来说，这是一个娱乐性的比赛）。最终的大尖峰发生在比赛结束、颁奖礼和持续了一段时间的庆祝期间（尽管流的停止时间大约在下午 6 点）。

2.7　小结

本章介绍了用 Twitter 数据实现的一些数据挖掘应用。我们讨论了如何用 Twitter 平台注册一个应用，以获得安全凭证并与 Twitter API 交互。还介绍了下载推文的不同方式，特别是用 REST 端点搜索已经发布的推文，以及用流 API 保持连接为打开状态，并搜集即将发布的推文。

当观察一条推文的结构时，我们发现一条推文远多于 140 个字符。实际上，它是一个包含很多信息的复杂对象。

在刚开始分析时，我们探讨了基于实体的频率分析。我们重点关注的是 Twitter 特有的话题标签，被用户广泛用于跟踪特定主题。我们还讨论了**自然语言处理**的很多方面，如分词和词项的归一化。正如我们所看到的，Twitter 中使用的语言并不遵循标准英语的规范，而是具有特定的特征，如话题标签、用户提及、URL、表情符号等。另一方面，我们发现，像任何其他具有相当规模的自然语言语料库一样，Twitter 中的语言遵循统计学中的 Zipf 定律。

在分析的最后，我们介绍了时间序列。观察用户如何与实时事件交互非常有意思，而时间序列分析是分析大数据集的一个强大的工具。

下一章也是关于 Twitter 的，但关注的重点是用户。我们想了解与这个社交网络连接的用户。

第 3 章

Twitter 用户、粉丝和社区

本章将继续介绍 Twitter 数据挖掘。上一章重点介绍了如何分析推文，现在我们将注意力转移到用户、用户间的联系，以及他们的互动。

本章包含如下主题：

- 如何下载给定用户的好友和粉丝列表
- 如何分析用户、互粉好友等群体间的联系
- 如何度量用户在 Twitter 上的影响力和参与度
- 聚类算法，以及如何使用 scikit-learn 聚集用户
- 网络分析，以及如何使用它在 Twitter 上挖掘对话
- 如何创建动态地图以显示推文的位置

3.1 用户、好友和粉丝

Twitter 和其他流行社交网络的主要区别之一是用户连接的方式。Twitter 上的关系不一定是双向的。用户可以选择订阅其他用户的推文，成为他们的粉丝（即关注他们），但这种关注行为不一定会得到回报（回粉）。这与 Facebook、LinkedIn 等其他社交网络的情况截然不同——在这些社交网络中，关系的建立必须得到双方的确认。

依照 Twitter 的术语，两类社交关系（我关注的人和关注我的人）有不同的名称。我关注的人称为**好友**，关注我的人称为我的**粉丝**。当一对用户的关系是双向的，通常描述为**互粉**。

3.1.1 回到 Twitter API

Twitter API 提供了几个端点来检索有关粉丝、好友和用户资料的信息。具体端点的选择由要解决的任务决定。

快速浏览一下开发者文档，会发现其中一些有趣的端点。

从单用户资料开始，明显可以使用的端点是 users/show：给定昵称或用户 ID，端点将检索用户的完整资料。此功能严重受到访问频率限制，因为每 15 分钟只能有 180 个请求，这意味着在 15 分钟的窗口中只有 180 份用户资料。鉴于此限制，只在需要检索特定的用户资料时，才应该使用该端点，而且该端点不适合批量下载。

可以用 followers/list 端点检索给定用户的粉丝列表，该功能在 Tweepy 中通过 API.followers()函数实现。与之类似，followers/list 端点允许通过 Tweepy 实现来检索给定用户的好友列表，该功能在 Tweepy 中通过 API.friends()实现。这些端点的主要问题是严格的频率限制：15 分钟窗口中只有 15 个请求，每个请求最多可提供 20 份用户资料。这意味着每 15 分钟最多可检索 300 份用户资料。

虽然这种方法对于拥有少量粉丝和好友的用户资料可行，但拥有数千名粉丝（对名人而言，可能会达到数百万）的用户资料并不少见。

此限制的解决方法基于更适合大批数据的其他端点的组合。followers/ids 端点可以针对每个请求返回 5000 个用户 ID。虽然在 15 分钟的窗口中也限制为 15 个请求，但很容易计算出可以检索到的最终用户 ID 数（每 15 分钟 75 000 个），比之前的限制好多了。

使用 followers/ids 端点后，我们拥有的是与用户的粉丝对应的用户 ID 列表，但还没有完整的资料。解决方案是，将这些用户 ID 作为 users/lookup 端口的输入。该端口最多可以输入 100 个 ID，提供相应的完整资料的列表作为输出。users/lookup 的频率限制设置为每 15 分钟 180 个，即每 15 分钟 18 000 份资料。这个数字有效地限制了下载次数。

如果想要下载一批好友的个人资料，解决方法是将 friends/ids 端点与上述 users/lookup 结合在一起。在频率限制方面，下载好友的 ID 时会遇到与下载粉丝 ID 时相同的限制。一个需要避免的小错误是将两个下载过程当作完全独立的；记住，users/lookup 是下载瓶颈。对于下载好友和粉丝个人资料的脚本来说，要考虑到这两个下载过程都需要 users/lookup 端点，因此，向 users/lookup 发起的请求总数是好友下载量和粉丝下载量的总和。对于既是好友又是粉丝的用户，只需要查找一次。

3.1.2 用户资料的结构

在详细了解如何下载大量粉丝和好友的个人资料前，我们先探究一下单个用户的情况来了解用户个人资料的结构。可以使用 users/show 端点，因为这是一个一次性的示例，所以不会达到频率限制（下一节将讨论批量下载）。

该端点可以在 Tweepy 中通过 API.get_user()函数实现。我们可以复用第 2 章中定义的鉴权代码（需要确认已经正确配置了环境变量）。在交互式解释器中输入以下命令。

```
>>> from twitter_client import get_twitter_client
>>> import json
>>> client = get_twitter_client()
>>> profile = client.get_user(screen_name="PacktPub")
>>> print(json.dumps(profile._json, indent=4))
```

以上代码比较直观。完成鉴权后，可以用一个 API 调用来下载用户资料。函数调用返回的对象是 `tweepy.models.User` 类的一个实例，第 2 章中介绍过，它是用于存储不同用户属性的包装器。Twitter 的原始 JSON 响应以一个 Python 字典的形式存储在`_json` 属性中，可以用`json.dumps()`和 `indent` 参数将该属性漂亮地打印到屏幕上。

我们将看到一个与以下代码段类似的 JSON 片段（为简洁起见，省略了一些属性）。


```
{
  "screen_name": "PacktPub",
  "name": "Packt Publishing",
  "location": "Birmingham, UK",
  "id": 17778401,
  "id_str": "17778401",
  "description": "Providing books, eBooks, video tutorials, and
    articles for IT developers, administrators, and users.",
  "followers_count": 10209,
  "friends_count": 3583,
  "follow_request_sent": false,
  "status": { ... },
  "favourites_count": 556,
  "protected": false,
  "verified": false,
  "statuses_count": 10802,
  "lang": "en",
  "entities": {
    "description": {
      "urls": []
    },
    "url": {
      "urls": [
        {
          "indices": [
            0,
            22
          ],
          "display_url": "PacktPub.com",
          "expanded_url": "http://www.PacktPub.com",
          "url": "http://t.co/vEPCgOu235"
        }
      ]
    }
  },
  "following": true,
  "geo_enabled": true,
  "time_zone": "London",
  "utc_offset": 0,
}
```

表 3-1 给出了我们可以找到的所有字段的详细描述。

表 3-1 用户资料属性及其描述

属性名称	描　述
_json	带有用户资料的 JSON 状态响应的字典
created_at	创建用户账户的 UTC 时间
contributors_enabled	表示**贡献者模式**为开启状态（极少为 true）的标记
default_profile	表示用户未改变资料主题的标记
description	描述用户资料的字符串
default_profile_image	表示用户没有自定义资料图片的标记
entities	URL 或描述中的实体列表
followers_count	粉丝的数量
follow_request_sent	表示是否有关注请求的标记
favourites_count	用户喜欢的推文数量
following	表示鉴权用户是否正在关注的标记
friends_count	朋友的数量
geo_enabled	表示**地理标签**为开启状态的标记
id	用户的唯一 ID，是一个较大的整数
id_str	用户的唯一 ID，是一个字符串
is_translator	表示用户为 Twitter 翻译者社区成员的标记
lang	用户的语言代码
listed_count	用户所属的公众用户列表的数量
location	用户声明的位置的字符串
name	用户的名字
profile_*	与用户资料相关的信息的数量（指以下与资料相关的属性列表）
protected	表示用户是否隐藏其推文的标记
status	这是最新推文中的嵌入对象（所有字段参考第 2 章）
screen_name	用户的昵称，即 Twitter 用户名
statuses_count	推文的数量
time_zone	用户声明的时区字符串
utc_offset	与 GMT/UTC 的偏差，单位为秒
url	用户提供的与资料相关的 URL
verified	表示是否已经认证用户的标记

与用户资料相关的属性如下所示。

- profile_background_color：这是用户为其背景选定的十六进制颜色代码值

- profile_background_tile：这是一个标记，表明显示 profile_background_image_url 时应该平铺
- profile_link_color：这是用户选定在 Twitter UI 中显示链接的十六进制颜色代码值
- profile_use_background_image：这是一个标记，表示用户想要采用自己上传的背景图片
- profile_background_image_url：这是一个基于 HTTP 的 URL，指向上传的背景图片
- profile_background_image_url_https：同上，但这是基于 HTTPS 的 URL
- profile_text_color：这是用户选定在 Twitter UI 中显示文本的十六进制颜色代码值
- profile_banner_url：这是一个基于 HTTPS 的 URL，指向用户上传的资料横幅
- profile_sidebar_fill_color：这是用户选定在 Twitter UI 中显示侧边栏背景的十六进制颜色代码值
- profile_image_url：这是一个基于 HTTPS 的 URL，指向用户的头像
- profile_image_url_https：同上，但这是基于 HTTPS 的 URL
- profile_sidebar_border_color：这是用户选定在 Twitter UI 中显示侧边栏边框的十六进制颜色代码值

接下来将介绍如何下载用户的好友和粉丝的资料。

3.1.3 下载好友和粉丝的资料

我们在前一节中讨论了相关端点，下面创建一个脚本，以一个用户名（昵称）作为输入，并下载他的完整信息、粉丝列表（包括完整资料）以及好友列表（包括完整资料）。

```
# Chap02-03/twitter_get_user.py
import os
import sys
import json
import time
import math
from tweepy import Cursor
from twitter_client import get_twitter_client

MAX_FRIENDS = 15000

def usage():
  print("Usage:")
  print("python {} <username>".format(sys.argv[0]))

def paginate(items, n):
  """为词项生成长度为 n 的块"""
  for i in range(0, len(items), n):
    yield items[i:i+n]

if __name__ == '__main__':
  if len(sys.argv) != 2:
```

```
    usage()
    sys.exit(1)
  screen_name = sys.argv[1]
  client = get_twitter_client()
  dirname = "users/{}".format(screen_name)
  max_pages = math.ceil(MAX_FRIENDS / 5000)
  try:
    os.makedirs(dirname, mode=0o755, exist_ok=True)
  except OSError:
    print("Directory {} already exists".format(dirname))
  except Exception as e:
    print("Error while creating directory {}".format(dirname))
    print(e)
    sys.exit(1)

  # 获取给定用户的粉丝
  fname = "users/{}/followers.jsonl".format(screen_name)
  with open(fname, 'w') as f:
    for followers in Cursor(client.followers_ids,
                            screen_name=screen_name).pages(max_pages):
      for chunk in paginate(followers, 100):
        users = client.lookup_users(user_ids=chunk)
        for user in users:
          f.write(json.dumps(user._json)+"\n")
      if len(followers) == 5000:
        print("More results available. Sleeping for 60 seconds to
              avoid rate limit")
        time.sleep(60)

  # 获取给定用户的好友
  fname = "users/{}/friends.jsonl".format(screen_name)
  with open(fname, 'w') as f:
    for friends in Cursor(client.friends_ids,
                          screen_name=screen_name).pages(max_pages):
      for chunk in paginate(friends, 100):
        users = client.lookup_users(user_ids=chunk)
        for user in users:
          f.write(json.dumps(user._json)+"\n")
      if len(friends) == 5000:
        print("More results available. Sleeping for 60 seconds to
              avoid rate limit")
        time.sleep(60)

  # 获取用户资料
  fname = "users/{}/user_profile.json".format(screen_name)
  with open(fname, 'w') as f:
    profile = client.get_user(screen_name=screen_name)
    f.write(json.dumps(profile._json, indent=4))
```

以上代码需要从命令行传入一个参数，即你想要分析的用户昵称。例如，可以用以下命令运行代码。

```
$ python twitter_get_user.py PacktPub
```

3

在撰写本书时，PacktPub 已经拥有 10 000 多名粉丝，所以由于 API 的接口访问限制，该代码的运行时间会超过 2 分钟。正如第 2 章中所说，该脚本用 `time.sleep()` 来减慢程序的运行速度，以避免触及接口访问频率限制。该 API 规定了传递给 `sleep()` 函数的秒数。

3.1.4　分析你的社会网络

下载好给定用户的好友和粉丝数据后，可以尝试对这些连接创建的网络结构做探索性分析。图 3-1 展示了一个虚构的用户网络示例，从第一人称的角度高亮显示了用户间的链接关系（即从标记为**我**的用户的角度，以第一人称视角描述这张图片）。

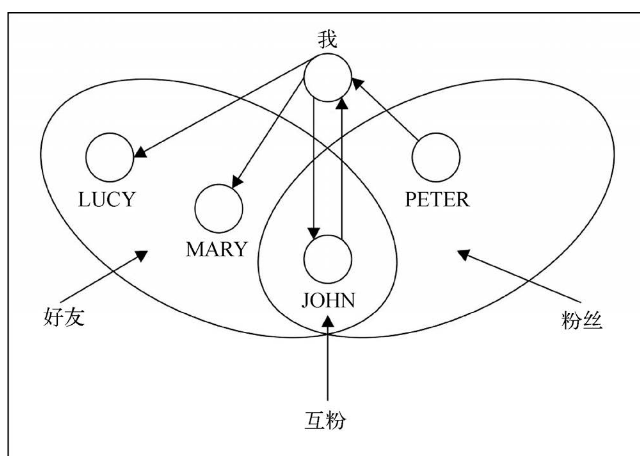


图 3-1　你的社会网络示例：好友、粉丝和互粉的好友

在这个示例中，我与 4 名用户连接：PETER 和 JOHN 关注了我（因此他们被标记为**粉丝**），而我关注了 LUCY、MARY 和 JOHN（因此他们被标记为**好友**）。JOHN 同时属于两个组：**好友**和**粉丝**的交集称作**互粉好友**。

我们并不能从图 3-1 中得知这 4 名用户之间是否存在连接关系。这是由我们在前一节中下载的数据决定的：我们只拥有给定用户的好友和粉丝信息，如果想要发现他们之间的连接，可以进一步迭代所有这些粉丝和好友的资料，并下载相关数据。

基于这些数据，我们拥有了粉丝和好友数量的基本统计值，现在可以回答以下基本问题。

- 谁是我的互粉好友？
- 我关注的哪些人没有回粉？
- 我没有回粉哪些粉丝？

以下代码读入前面下载的 JSONL 文件并计算相关的统计值，以回答上述问题。

```
# Chap02-03/twitter_followers_stats.py
import sys
import json
from random import sample
import time

def usage():
  print("Usage:")
  print("python {} <username>".format(sys.argv[0]))

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  screen_name = sys.argv[1]
  followers_file = 'users/{}/followers.jsonl'.format(screen_name)
  friends_file = 'users/{}/friends.jsonl'.format(screen_name)
  with open(followers_file) as f1, open(friends_file) as f2:
    t0 = time.time()
    followers = []
    friends = []
    for line in f1:
      profile = json.loads(line)
      followers.append(profile['screen_name'])
    for line in f2:
      profile = json.loads(line)
      friends.append(profile['screen_name'])
    t1 = time.time()
    mutual_friends = [user for user in friends if user in followers]
    followers_not_following = [user for user in followers if user not in friends]
    friends_not_following = [user for user in friends if user not in followers]
    t2 = time.time()
    print("----- Timing -----")
    print("Initialize data: {}".format(t1-t0))
    print("Set-based operations: {}".format(t2-t1))
    print("Total time: {}".format(t2-t0))
    print("----- Stats -----")
    print("{} has {} followers".format(screen_name, len(followers)))
    print("{} has {} friends".format(screen_name, len(friends)))
    print("{} has {} mutual friends".format(screen_name, len(mutual_friends)))
    print("{} friends are not following {} back".format(len(friends_not_following),
      screen_name))
    print("{} followers are not followed back by {}".format(len(followers_not_
      following), screen_name))

    some_mutual_friends = ', '.join(sample(mutual_friends, 5))
    print("Some mutual friends: {}".format(some_mutual_friends))
```

运行以上代码需要从命令行传入一个用户名参数。可以用以下命令运行该脚本。

```
$ python twitter_followers_stats.py PacktPub
```

输出结果如下所示。

```
PacktPub has 10209 followers
PacktPub has 3583 friends
PacktPub has 2180 mutual friends
1403 friends are not following PacktPub back
8029 followers are not followed back by PacktPub
```

用于处理好友和粉丝的数据类型是常规的 Python `list()`，填充的元素是用户名（准确地说是 `screen_name`）。该代码读入两个 JSONL 文件，并分别将昵称加入列表。三个列表推导式用于计算不同的统计指标。



Python 中的列表推导式

Python 支持名为“列表推导式”的概念，它是一种将一个列表（或任何可迭代的数据结构）优雅地转换为另一个列表的方式。在这个过程中，自定义函数会有条件地筛选并转换元素，如下所示。

```
>>> numbers = [1, 2, 3, 4, 5]
>>> squares = [x*x for x in numbers]
>>> squares
[1, 4, 9, 16, 25]
```

前面的列表推导式与以下代码等价。

```
>>> squares = []
>>> for x in numbers:
...   squares.append(x*x)
...
>>> squares
[1, 4, 9, 16, 25]
```

列表推导式的一个优点是可以用普通英语读懂，因此代码的可读性很强。

推导式不局限于列表，实际上，它还可以用于创建字典。

实现方面还需要考虑以下几点。首先，JSONL 文件将包括唯一的资料：每个粉丝（或好友）只会列举一次，因此不会有重复的项。其次，当计算前面的统计值时，每一项的顺序不相关。因此，实际上我们做的是基于集合的操作（这个示例中是交集和差集）。

可以用 `set()` 来重构我们的代码，以实现基本的统计。代码的主要变化是需要重新导入数据，并计算互粉好友、单方关注的好友和单方关注的粉丝。

```
with open(followers_file) as f1, open(friends_file) as f2:
  followers = set()
  friends = set()
  for line in f1:
    profile = json.loads(line)
    followers.add(profile['screen_name'])
  for line in f2:
    profile = json.loads(line)
    friends.add(profile['screen_name'])
  mutual_friends = friends.intersection(followers)
```

```
  followers_not_following = followers.difference(friends)
  friends_not_following = friends.difference(followers)
```

该代码的输出结果与使用列表的输出结果相同。相较于列表来说，使用集合的主要优势在于计算的复杂度：对包含（即检查 `item in list` 或 `item in set`）这样的操作来说，列表的运行时间是线性的，而集合的运行时间是常量。包含用于构建 `mutual_friends`、`followers_not_following` 和 `friends_not_following`，因此这个简单的优化将显著改善代码的运行时间。对于拥有很多粉丝/好友的分析来说，这个区别更加明显，因为列表的运行时间是线性增长的。



计算的复杂度

前面我们使用了**线性时间**、**常量时间**、**线性复杂度**等词语。计算复杂度是计算机科学中非常重要的一个概念，它主要考虑运行算法需要占用的资源。这里我们将介绍时间复杂度和运行算法需要占用的时间，它是一个与输入大小相关的函数。当一个算法以线性时间运行时，对于较大的输入，其运行时间随输入的扩大呈线性增长。时间复杂度的数学表示是 `O(n)`（也称作大 O 表示），其中 `n` 是输入的大小。对于以常数时间运行的算法来说，输入的大小并不影响运行时间。这种情况下，时间复杂度表示为 `O(1)`。

总的来说，理解不同操作和数据结构的复杂度是开发高效程序的关键步骤，因为这对系统的性能有较大影响。

至此，你可能会对 NumPy 等库的使用感兴趣。第 1 章中介绍过，NumPy 为类似数组的数据结构提供了快速高效的处理，并极大提升了简单列表的性能。虽然 NumPy 对速度进行过优化，但这里使用 NumPy 并不会显著提升性能，因为包含操作的计算开销与列表相同。使用 NumPy 重构上述代码产生的结果如下所示。

```
with open(followers_file) as f1, open(friends_file) as f2:
  followers = []
  friends = []
  for line in f1:
    profile = json.loads(line)
    followers.append(profile['screen_name'])
  for line in f2:
    profile = json.loads(line)
    friends.append(profile['screen_name'])
  followers = np.array(followers)
  friends = np.array(friends)
  mutual_friends = np.intersect1d(friends,
                                  followers,
                                  assume_unique=True)
  followers_not_following = np.setdiff1d(followers,
                                         friends,
                                         assume_unique=True)
  friends_not_following = np.setdiff1d(friends,
                                       followers,
                                       assume_unique=True)
```

正如第 1 章中所说，这段代码假设我们已经用 np 别名导入了 NumPy。这个库也提供了类似集合的操作 intersect1d 和 setdiff1d，但其底层的数据结构仍然是类似数组的对象（如列表）。这些函数的第三个参数 assume_unique 用于输入数组具有唯一元素时，如以上示例所示。这样做可以加速计算，但底线仍然相同：当粉丝/好友数众多时，集合会更快地执行这些操作。如果粉丝/好友数量很少，那么这些操作在性能上的差别不会很明显。

本书中的源代码提供了三种实现，分布在 twitter_followers_stats.py、twitter_followers_stats_set.py 和 twitter_followers_stats_numpy.py，并且其中包含了运行时间的计算，因此你可以用不同的数据来做实验。



对代码重构这个插曲做个总结。这里想表达的主要思想是，对于正在处理的操作，需要选择最合适的数据结构。

3.1.5　度量影响力和参与度

社会媒体领域最常提及的一个角色是神秘的**影响者**，这种角色造成了最近市场营销策略的范式转变，即以关键个人而不是整个市场为目标。

影响者通常是社区中的活跃用户。在 Twitter 中，影响者会发布人们关心的很多话题。影响者和社区中的很多其他用户都存在着关注和被关注的连接关系。通常，影响者通常也被视为其所在领域的专家，并受到其他用户信任。

以上描述解释了为什么影响者对于近期的营销趋势有较大影响——影响者可以增强人们对特定产品或品牌的意识，甚至成为产品或品牌的推荐人，帮助其获得众多支持者。

不论你的主要兴趣是 Python 编程还是品尝红酒，不管你的社会网络是大是小，你都应该知道谁是你的社交圈中的影响者：一个好友、一位熟人，或互联网上随机的陌生人——你十分信任和看重他的意见，因为他具有特定领域的专业知识。

一个不同但相关的概念是**参与度**。用户参与度（或客户参与度）是对特定产品或服务的响应的评估。在社会媒体中，有些内容的创建目的是增加公司网站或者电子商务的流量。度量参与度非常重要，因为它可以帮助我们定义和理解营销策略，以便最大化与所在社会网络的互动，最终带来生意。在 Twitter 中，用户参与是指转发或者喜欢某条推文，这些行为增强了原始推文的可见性。

接下来我们将介绍社会媒体分析中一些非常有趣的方面，其中包括度量影响力和参与度。在 Twitter 中，一种很自然的想法是将特定网络的影响力与其用户数量关联起来。直观来看，拥有大量粉丝意味着用户可以连接到更多的人，但这并不能说明一条推文是如何被感知的。

以下代码比较了两份用户资料的一些统计值。

```
# Chap02-03/twitter_influence.py
import sys
import json

def usage():
  print("Usage:")
  print("python {} <username1> <username2>".format(sys.argv[0]))

if __name__ == '__main__':
  if len(sys.argv) != 3:
    usage()
    sys.exit(1)
  screen_name1 = sys.argv[1]
  screen_name2 = sys.argv[2]
```


从命令行读入两个昵称后，可以分别为这两位用户构建粉丝列表（包括粉丝的数量），以计算可连接的用户数量。

```
  followers_file1 = 'users/{}/followers.jsonl'.format(screen_name1)
  followers_file2 = 'users/{}/followers.jsonl'.format(screen_name2)
  with open(followers_file1) as f1, open(followers_file2) as f2:
    reach1 = []
    reach2 = []
    for line in f1:
      profile = json.loads(line)
      reach1.append((profile['screen_name'],
                     profile['followers_count']))
    for line in f2:
      profile = json.loads(line)
      reach2.append((profile['screen_name'],
                     profile['followers_count']))
```

然后从两份用户资料导入一些基本统计值（粉丝数量和状态数量）。

```
profile_file1 = 'users/{}/user_profile.json'.format(screen_name1)
profile_file2 = 'users/{}/user_profile.json'.format(screen_name2)
with open(profile_file1) as f1, open(profile_file2) as f2:
  profile1 = json.load(f1)
  profile2 = json.load(f2)
  followers1 = profile1['followers_count']
  followers2 = profile2['followers_count']
  tweets1 = profile1['statuses_count']
  tweets2 = profile2['statuses_count']

sum_reach1 = sum([x[1] for x in reach1])
sum_reach2 = sum([x[1] for x in reach2])
avg_followers1 = round(sum_reach1 / followers1, 2)
avg_followers2 = round(sum_reach2 / followers2, 2)
```

接着导入两个用户的时间线，并观察他们的推文被喜欢或转发的次数。

```
timeline_file1 = 'user_timeline_{}.jsonl'.format(screen_name1)
timeline_file2 = 'user_timeline_{}.jsonl'.format(screen_name2)
```

```
with open(timeline_file1) as f1, open(timeline_file2) as f2:
  favorite_count1, retweet_count1 = [], []
  favorite_count2, retweet_count2 = [], []
  for line in f1:
    tweet = json.loads(line)
    favorite_count1.append(tweet['favorite_count'])
    retweet_count1.append(tweet['retweet_count'])
  for line in f2:
    tweet = json.loads(line)
    favorite_count2.append(tweet['favorite_count'])
    retweet_count2.append(tweet['retweet_count'])
```

以上数量聚合为喜欢和转发的平均数量，而且都是绝对值且基于粉丝数量。

```
avg_favorite1 = round(sum(favorite_count1) / tweets1, 2)
avg_favorite2 = round(sum(favorite_count2) / tweets2, 2)
avg_retweet1 = round(sum(retweet_count1) / tweets1, 2)
avg_retweet2 = round(sum(retweet_count2) / tweets2, 2)
favorite_per_user1 = round(sum(favorite_count1) / followers1, 2)
favorite_per_user2 = round(sum(favorite_count2) / followers2, 2)
retweet_per_user1 = round(sum(retweet_count1) / followers1, 2)
retweet_per_user2 = round(sum(retweet_count2) / followers2, 2)
print("----- Stats {} -----".format(screen_name1))
print("{} followers".format(followers1))
print("{} users reached by 1-degree
      connections".format(sum_reach1))
print("Average number of followers for {}'s followers:
      {}".format(screen_name1, avg_followers1))
print("Favorited {} times ({} per tweet, {} per
      user)".format(sum(favorite_count1), avg_favorite1,
      favorite_per_user1))
print("Retweeted {} times ({} per tweet, {} per
      user)".format(sum(retweet_count1), avg_retweet1,
      retweet_per_user1))
print("----- Stats {} -----".format(screen_name2))
print("{} followers".format(followers2))
print("{} users reached by 1-degree
      connections".format(sum_reach2))
print("Average number of followers for {}'s followers:
      {}".format(screen_name2, avg_followers2))
print("Favorited {} times ({} per tweet, {} per
      user)".format(sum(favorite_count2), avg_favorite2,
      favorite_per_user2))
print("Retweeted {} times ({} per tweet, {} per
      user)".format(sum(retweet_count2), avg_retweet2,
      retweet_per_user2))
```

以上脚本从命令行传入两个参数，并假设已经下载好数据。对于这两个用户，我们需要他们的粉丝数据（用 `twitter_get_user.py` 脚本下载）及其各自的时间线（用第 2 章中的 `twitter_get_user_timeline.py` 下载）。

以上代码有些冗长，因为对两份用户资料计算了相同的操作，并且将所有的东西打印到了终

端。我们可以将其分解为不同部分。

首先，研究一下粉丝的粉丝。这将提供与给定用户直接连接的那部分网络的信息。换句话说，它可以回答以下问题：如果所有粉丝都转发我的推文，会有多少用户可以看到？为了回答上述问题，我们读入 users/<user>/followers.jsonl 文件并获取一个元组列表，其中每个元组表示其中一个粉丝的(screen_name, followers_count)信息。这里保留昵称对后续找出谁的粉丝数量最多非常有用（并没有在代码中直接计算，可以用 sorted()函数轻松生成）。

第二步，从 users/<user>/user_profile.json 文件中读入用户资料，以获得有关粉丝和推文总量的信息。基于目前收集到的数据，我们可以计算一度分隔（即粉丝的粉丝）内可以连接到的用户总数，以及粉丝的粉丝的平均数量。可以通过以下代码计算。


```
sum_reach1 = sum([x[1] for x in reach1])
avg_followers1 = round(sum_reach1 / followers1, 2)
```

第一行代码用一个列表推导式来迭代前面提到的元组列表，第二行代码是一个简单的算术求平均，并四舍五入保留两位小数。

代码的第三部分从第 2 章生成的 user_timeline_<user>.jsonl 文件读入用户时间线，并收集有关每条推文转发和喜欢的信息。综合所有信息后，可以计算出一位用户的推文被转发和喜欢的次数，以及相对于每条推文和每个粉丝来说，推文被转发和喜欢的平均次数。

为了提供示例，我将做一个满足自己虚荣心的分析，对我的账户@marcobonzanini 和 Packt 出版社的账户进行比较。

```
$ python twitter_influence.py marcobonzanini PacktPub
```

该代码生成的输出如下所示。

```
----- Stats marcobonzanini -----
282 followers
1411136 users reached by 1-degree connections
Average number of followers for marcobonzanini's followers: 5004.03
Favorited 268 times (1.47 per tweet, 0.95 per user)
Retweeted 912 times (5.01 per tweet, 3.23 per user)
----- Stats PacktPub -----
10209 followers
29961760 users reached by 1-degree connections
Average number of followers for PacktPub's followers: 2934.84
Favorited 3554 times (0.33 per tweet, 0.35 per user)
Retweeted 6434 times (0.6 per tweet, 0.63 per user)
```

正如你看到的，粉丝的数量没有可比性，Packt 出版社的粉丝量约是我的 35 倍。这个分析结果有趣的部分在后面：当分析转发和喜欢的平均数时，粉丝对我所分享内容的参与度明显高于 PacktPub。这能充分表明我是一个影响者，而 PacktPub 不是吗？显然不能。我们可以发现，我的推文大多聚焦于特定主题（Python 和数据科学），因此我的粉丝已经对我发布的主题感兴趣。而

Packt 出版社发布的内容非常多样，包含很多技术。这种多样性也反映在 PacktPub 的粉丝分布上，其粉丝包括开发者、设计师、科学家、系统管理员等。因此，PacktPub 的每条推文只会被一小部分粉丝转发。

3.2　挖掘粉丝

在现实世界中，社交群体是指具备共同条件的一组人。这个广泛的定义包括在同一地理位置的人、拥有相同政治或宗教信仰的人、拥有相同兴趣（如阅读）的人，等等。

社交群体的概念也是社会媒体平台的核心。虚拟环境中群体的界限比现实世界中更模糊，因为地理位置不再是决定性因素。就像在面对面场景中一样，当有共同兴趣或条件的人开始交互时，社会媒体平台上自然就形成了社群。

可以根据成员是否意识到自己是某社群的成员来区别不同类型的社群。在**显式**社群中，成员和非成员都非常清楚他们是否属于该社群，并知道社群中的其他成员都是谁。社群成员间的互动会比与非成员的互动更加频繁。

另一方面，**隐式**社群的存在并没有得到明确公认。这种社群的成员可能具有相同的兴趣，但并没有明显和强烈的联系。

本节将分析用户数据，并对一组用户资料进行分类，从而分析出用户的共同兴趣和条件。

聚类（即聚类分析）是一种机器学习技术，用于对数据项分组，同一组（即**簇**）内对象间的相似度会高于不同组间对象的相似度。聚类属于**无监督学习**技术，也就是说，我们处理的对象并没有被打过标签。无监督学习的目的是发现数据的隐含结构。

常见的一个聚类应用场景是市场研究。例如，按照相似的行为/兴趣对一群客户进行分组，从而对他们进行不同的产品推广或者营销推广。社会网络分析是聚类可以发挥重大作用的另一个领域，因为聚类可用于识别一大群人的社群结构。

聚类并不是一种具体的算法，而是可以用不同算法实现的一个任务。我们的分析选择的算法是 **k-means**，它是聚类最常用的方法之一。k-means 是一个方便的选项，因为它易于理解和实现，而且与其他算法比较，具有更高的计算效率。

k-means 需要两个参数：n 维空间中的大量输入向量，它表示我们想要聚类的对象；我们期望将数据分成的簇的数量 K。因为使用的是无监督学习，所以我们不需要拥有数据的正确类别标签。具体来说，我们甚至不知道簇的正确（或理想的）数量。这个信息在聚类分析中非常重要，但现在并不需要太过担心这个问题。

上一段中定义了我们希望聚类为 n 维空间中向量的对象。简单来说，这意味着每一份用户资

料将表示为由 n 个数值元素（即**特征**）构成的一个向量。

定义这些特征的方法基于用户的描述，即用户提供的有关自己兴趣或职位的文本信息。将文本描述转换为特征向量的过程称为向量化。scikit-learn 中提供了 k-means 算法和其他向量化工具的实现。scikit-learn 是一个 Python 的机器学习工具，我们在第 1 章中介绍过。

向量化过程包括将用户描述分解为词项，然后为每个词项分配特定的**权重**。本例中采用的权重分配方案是常用的 TF-IDF 方法，它是**词频**（term frequency，TF）和**逆文档频率**（inverse document frequency，IDF）的组合。TF 是一个局部统计，表示单词出现在文档中的频率。而 IDF 是一个全局统计，表示单词在文档集合中的稀缺性。这些统计值常用于搜索引擎和不同的文本挖掘应用中，度量了单词在给定内容中的重要程度。

TF-IDF 提供了一个简单直观的数值权重：如果一个单词在一篇文档中频繁出现，但在整个文档集合中较少出现，那么它可能对这篇文档而言具有较强的代表性，因此应该赋予较高的权重。在这个应用中，我们将用户描述当作文档，并用 TF-IDF 统计值将这些用户描述表示为向量元素。获得了 n 维向量表示后，可以用 k-means 算法计算这些向量的相似度。

以下脚本用到了 scikit-learn，具体而言是 `TfidfVectorizer` 和 `KMeans` 类。

```
# Chap02-03/twitter_cluster_users.py
import sys
import json
from argparse import ArgumentParser
from collections import defaultdict
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

def get_parser():
  parser = ArgumentParser("Clustering of followers")
  parser.add_argument('--filename')
  parser.add_argument('--k', type=int)
  parser.add_argument('--min-df', type=int, default=2)
  parser.add_argument('--max-df', type=float, default=0.8)
  parser.add_argument('--max-features', type=int, default=None)
  parser.add_argument('--no-idf', dest='use_idf', default=True,
                      action='store_false')
  parser.add_argument('--min-ngram', type=int, default=1)
  parser.add_argument('--max-ngram', type=int, default=1)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  if args.min_ngram > args.max_ngram:
    print("Error: incorrect value for --min-ngram ({}): it can't
          be higher than --max-value ({})".format(args.min_ngram,
          args.max_ngram))
    sys.exit(1)
```

```
with open(args.filename) as f:
  # 加载数据
  users = []
  for line in f:
    profile = json.loads(line)
    users.append(profile['description'])
  # 创建向量化器
  vectorizer = TfidfVectorizer(max_df=args.max_df,
                               min_df=args.min_df,
                               max_features=args.max_features,
                               stop_words='english',
                               ngram_range=(args.min_ngram,
                                            args.max_ngram),
                               use_idf=args.use_idf)
  # 拟合数据
  X = vectorizer.fit_transform(users)
  print("Data dimensions: {}".format(X.shape))
  # 执行聚类
  km = KMeans(n_clusters=args.k)
  km.fit(X)
  clusters = defaultdict(list)
  for i, label in enumerate(km.labels_):
    clusters[label].append(users[i])
  # 打印这个簇的前 10 个用户描述
  for label, descriptions in clusters.items():
    print('---------- Cluster {}'.format(label))
    for desc in descriptions[:10]:
      print(desc)
```

TfidfVectorizer 类提供了一些选项来配置计算向量的方式，我们将用 ArgumentParser（Python 标准库的一部分）从命令行获得这些选项。get_parser() 函数定义的参数如下所示。

- --filename：我们想要分析的 JSONL 文件名的路径
- --k：簇的数量
- --min-df：一个特征的最小文档频率（默认是 2）
- --max-df：一个特征的最大文档频率（默认是 0.8）
- --max-features：这是特征的最大数量（默认是 None）
- --no-idf：该标记表示我们是否希望取消 IDF 权重，只使用 TF（默认使用 IDF）
- --min-ngram：这是抽取的 n-gram 的最小边界值（默认值是 1）
- --max-ngram：这是抽取的 n-gram 的最大边界值（默认值是 1）

可以用以下命令运行上述脚本：

```
$ python twitter_cluster_users.py \
  --filename users/marcobonzanini/followers.jsonl \
  --k 5 \
  --max-features 200 \
  --max-ngram 3
```

必备参数是--filename 和--k，前者表示我们要分析的文件名，后者是簇的数量。其他参数都是可选的，并用于定义 TfidfVectorizer 如何为 KMeans 创建向量。

通过显式使用--max-features 或者用--min-df 和--max-df 来指定一个特征的期望文档频率范围，我们可以限制 TfidfVectorizer 抽取的特征数量。最小文档频率默认是 2，这意味着，如果特征只在少于 2 篇文档中出现，那么就会被忽略。另一方面，最大文档频率默认是 0.8，这意味着，如果特征在 80%以上的文档中出现，那么就会被忽略。根据数据集的大小，我们可以或多或少保守地确定这些值。基于文档频率排除这些特征的目的是避免计算不具代表性的特征。此外，用--max-features 限制特征的数量可以加速计算，因为输入更小。指定这个选项后，就可以确定最频繁的特征（取决于它们的文档频率）。

与其他参数不同，--no-idf 选项不用于指定特定的值，而是取消 IDF 的计算（这意味着只用 TF 计算特征权重）。我们需要将一个变量名为目的地的变量（dest=use_idf）和给出 store_false 参数（当该参数未给出时，其默认值时 True）时需要执行的动作传递给参数解析器，这样就可以指定--no-idf 参数的行为。IDF 通常用于缩小在文档集合中经常出现、因而不具有任何特定辨别力的单词的权重。虽然在很多应用中，在权重函数中采用 IDF 是一个绝佳选择，但在探索性数据分析的步骤中，观察其实际效果仍然非常重要。因此，关闭该功能的选项只是我们工具箱中的一个额外工具。

最后两个参数允许我们采用 n-gram 作为特征，而不是单个单词。通常来说，一个 n-gram 是 *n* 项的连续序列。在我们的应用中，我们将一段文本切分为单词序列。单个单词称为 unigram（n=1 的 n-gram）。其他常用的 n-gram 还有 bigram（n=2）和 trigram（n=3），是否采用更大的 gram 取决于实际的应用。例如，句子 the quick brown fox jumped over the lazy dog 的 bigram 结果为：the quick、quick brown、brown fox、fox jumped，等等。

使用 n-gram 而不是 unigram 的好处是能捕捉到短语。思考以下两个句子：

- He’s a scientist, but he doesn’t like data
- He works as a data scientist

如果只用 unigram，那么在这两个句子中都会捕获到 data 和 scientist 项，尽管这两项在两个句子中的使用方式完全不同，但如果使用 bigram，则可以捕获到短语 data scientist，它的含义与两个单独的单词不同。默认情况下，n-gram 的下界和上界的值设置为 1。也就是说，如果没有特别指定，我们使用的是 unigram。

我们来总结一下如何配置 TfidfVectorizer。首先，它使用 stop_words 参数定义从英语词汇表中捕获的停用词。它是一个常见英语单词（如 the 和 and）列表，这些单词本身并不带有特定意义，几乎用于每段文本中。在常见英语中，停用词可能已经被 max_df 选项过滤掉了（因为它们的文档频率接近 100%），Twitter 用户并不经常使用流畅的常见英语，而是用一组关键词来

表示他们的兴趣，以克服推文的字符限制。TfidfVectorizer 也允许使用 stop_words 属性传入我们自定义的停用词列表。

运行前面的代码会输出有趣的结果。为简洁起见，图 3-2 展示了部分输出结果。

```
# ---------- Cluster 0
PhD Student #FutureOfNews, #TextMining, Topic Detection & Tracking,
Summarization, Information Retrieval, mostly with Breaking News
Data
PhD candidate - How and what can search engines learn from their
users?
PhD candidate. Information Retrieval. Data Mining. Open Source.
PhD Candidate | Semantic Search in eDiscovery | Information
Retrieval, Text Mining
# ---------- Cluster 1
Co-fundador de Viz Analytics. Diplomado en Ingeniería Informatica
de Gestión y con un Master Universitario en Inteligencia
Artificial.
Lenguas, comunicación y nuevos medios. No tuitear en caso de
incendio.
Lic. en Administración mención informática, promotor del software
libre y las oportunidades que brinda.
# ---------- Cluster 2
I am part time web-developer and I love python
boiling down to Python and Infosec nowadays. Any advices?
Python Padawan, Django noob.
Sherlock of #Data, Love #machinelearning #python
# ---------- Cluster 3
Data Scientist
Data scientist in the making
Data scientist, interested in text analysis, social networks,
artificial and natural intelligence, data, complexity, illusions...
Data Scientist
Wanna be a Data Scientist: using data for good
```

图 3-2　部分输出结果

为简洁起见，对输出结果做了截断，突出显示了 k-means 算法的一些有趣行为。

粉丝的第一个簇是一群学者和博士生，他们的主要研究方向是文本分析和信息检索。可以看到，两份资料描述中的 PhD candidate 和 Information Retrieval 短语是非常明显的匹配用户相似度的证据。

第二个簇由西班牙语用户组成。k-means 和 TfidfVectorizer 并没有关于多语言的知识。与语言相关的唯一方面是使用常见英语单词的停用词。k-means 能够智能到识别语言，并用这个信息将用户分组吗？注意，我们创建的向量基于词袋（更准确说是 n-gram 词袋）表示，因此相似度只是基于单词和 n-gram 的简单重合。即便不懂西班牙语，我们仍然能发现几个在不同资料中经常出现的单词（如 en、y 和 de）。这些单词可能在整个粉丝（大多是英语使用者）集合中非常稀少，因此，其重要程度体现在很高的 IDF 值。虽然它们也碰巧是西班牙语的停用词，但因为这里使用的停用词列表是针对英语的，所以它们仍然保留为向量特征。除了语言特性，这些资料之间的连接可能非常松散。第一个和第三个资料提到了一些与计算机科学相关的项。而第二个资料提到的是“语言、沟通和新媒体。如果发生火灾，不要发推文”。与其他资料的连接就非常模糊了，不过这仍然是一个有趣的结果。

与第一个簇相似，第三个和第四个簇可以较好地自我说明且非常一致：簇 3 主要由 Python 开发者组成，簇 4 由数据科学家组成。

因为 k-means 的初始值是任意的，所以即使每次用相同参数运行同样的程序也并不能确保可以得到同样的结果。如果感兴趣，我建议你尝试用不同的参数值（如使用更多或更少的特征，用 unigram 和更大的 n-gram 对比，用较小范围的文档频率和更大范围的文档频率对比等）运行程序，并观察这些参数如何影响算法的行为。

在讨论 k-means 之初，我们介绍过簇的数量 K 和数据是该算法的主要输入。找到给定数据集的最佳 K 值仍然是一个学术研究问题，而且与如何执行实际的聚类不同。当然，也有很多其他方法可以实现上述目的（参见维基百科词条"Determining the number of clusters in a data set"）。

最简单的方法是拇指法则，其做法是将 K 值设置为 $n/2$ 的平方根，其中 n 是数据集中对象的个数。其他方法会涉及簇内的一致性度量，如肘部法则或 Silhouette 方法。

3.3　挖掘对话

前面主要介绍了用户资料，以及他们如何通过粉丝/好友关系显式连接。本节将分析一种不同的交互类型——对话。在 Twitter 中，用户可以发布一条推文来对一段内容进行**回复**。当两位及更多用户参与这一过程时，一个对话就展开了。

图 3-3 展示了一个对话，表示为一个网络。网络中的每个节点都是一条推文（由其 ID 唯一标识），而每条边表示一个**回复**关系。

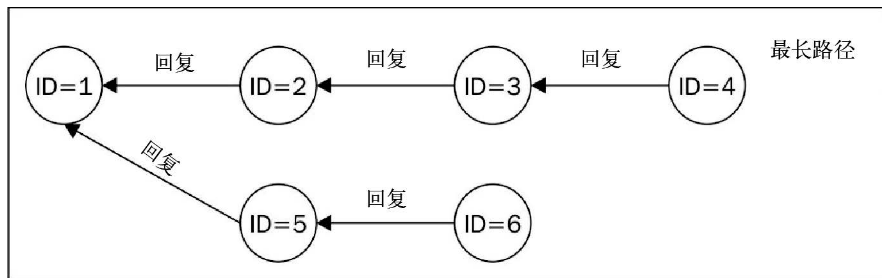


图 3-3　表示为网络的对话示例

这种关系有明确的方向，因为它是单向的（父子关系）。例如，如果推文 2 是对推文 1 的回复，那么推文 1 就不会是对推文 2 的回复。这种关系的排序总是唯一的，也就是说，给定推文只能是对一条推文的回复（但多条推文可对同一条推文进行回复，因此这是一对多关系）。此外，不允许出现循环（例如，如果一个关系序列是 1 到 2、2 到 3，那么 3 到 1 是不可能发生的）。因此，用于表示这种关系的图是有向无环图。更准确地说，这里表示的图的类型通常称作**有向树**。

虽然图 3-3 没有明确描述，但我们应该注意到回复关系是受时间限制的，即只能回复已经发布的推文。

通过用图表示推文和回复，可以用图的性质和算法来挖掘对话。

例如，一个节点的**度**是图中该节点的子节点数量。从对话的角度看，节点的度对应的是该节点收到的回复的数量。在图 3-3 的示例中，有两个节点直接与节点 1 相连，因此节点 1 的度是 2。节点 2、节点 3 和节点 5 都只与一个节点与其相连，因此它们的度都为 1。最后，节点 4 和节点 6 没有与之相连的节点，因此度为 0。它们也称作树的**叶子节点**，表示对话的结束。另一方面，对话的起点，即节点 1 也称作树的**根节点**。

图论中的一个基本概念是**路径**，它是连接节点的边的序列。给定一个推文序列的图表示，查找一条路径等于跟随一个对话。一个非常有趣的问题是查找最大长度的路径，也称作最长路径。

为了将这些图的概念应用于我们的 Twitter 数据，可以使用第 1 章中介绍过的 NetworkX 库，它提供了图数据结构的高效计算，以及一个非常简单的接口。以下脚本以推文的 JSONL 文件作为输入，然后生成前面介绍过的一个有向图。

```
# Chap02-03/twitter_conversation.py
import sys
import json
from operator import itemgetter
import networkx as nx

def usage():
  print("Usage:")
  print("python {} <filename>".format(sys.argv[0]))

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  fname = sys.argv[1]
  with open(fname) as f:
    graph = nx.DiGraph()
    for line in f:
      tweet = json.loads(line)
      if 'id' in tweet:
        graph.add_node(tweet['id'],
                       tweet=tweet['text'],
                       author=tweet['user']['screen_name'],
                       created_at=tweet['created_at'])
        if tweet['in_reply_to_status_id']:
          reply_to = tweet['in_reply_to_status_id']
          if tweet['in_reply_to_status_id'] in graph \
          and tweet['user']['screen_name'] !=
            graph.node[reply_to]['author']:
            graph.add_edge(tweet['in_reply_to_status_id'],
```

```
                          tweet['id'])
    # 打印一些基本统计值
    print(nx.info(graph))
    # 找出回复量最多的推文
    sorted_replied = sorted(graph.degree_iter(),
                            key=itemgetter(1),
                            reverse=True)
    most_replied_id, replies = sorted_replied[0]
    print("Most replied tweet ({} replies):".format(replies))
    print(graph.node[most_replied_id])
    # 找出最长的对话
    print("Longest discussion:")
    longest_path = nx.dag_longest_path(graph)
    for tweet_id in longest_path:
      node = graph.node[tweet_id]
      print("{} (by {} at {})".format(node['tweet'],
                                      node['author'],
                                      node['created_at']))
```


可以用以下命令运行上述脚本。

```
$ python twitter_conversation.py <filename>
```

在这段代码中，我们用 `nx` 别名导入 NetworkX，第 1 章中介绍过这一点。首先，初始化一个用 `nx.DiGraph` 类实现的空的有向图。输入的 JSONL 文件的每一行表示一条推文，我们将对这个文件进行迭代，将每条推文作为一个节点加入图中。`add_node()` 函数的作用就是加入节点，它的必备参数是节点 ID，此外，可选关键词参数的个数用于提供节点的额外属性。在示例中，我们将使用作者的昵称、推文的完整文本和推文的创建时间。

创建好节点后，我们将查看该推文是否为另一条推文的回复。如果是，我们可能想在两个节点间添加一条边。在添加边之前，需要先确认被回复的节点已存在于图中，这样可以避免图中出现对不在数据集中的推文的回复（如在我们用流 API 采集数据前发布的推文）。如果尝试连接的节点不在图中，`add_edge()` 函数会将其添加到图中，但除了其 ID，我们不知道其他属性的任何信息。还需要确认推文的作者与回复的作者是否为不同的人。这是因为 Twitter UI 自动将对话中的推文分组，但一些用户是对一个实时事件进行评论，或者通过对自己的推文回复一个超过 140 字符的评论来轻松创建一个多推文线程。这虽然是个不错的功能，但不是对话（事实上正相反），因此需要忽略这些线程。如果想发现自言自语，可以修改代码来实现相反的功能：只在推文的作者和回复的作者相同时添加边。

构建好图后，先打印一些由 `nx.info()` 函数提供的基本统计值。然后识别具有最多回复量的推文（即拥有最大度的节点）以及最长的对话（即最长路径）。

由于 `degree_iter()` 函数返回的是 `(node, degree)` 元组序列的迭代器，我们将用 `itemgetter()` 函数按照度的逆序进行排序。排序后的第一项就是拥有回复数量最多的推文。

查找最长对话的解决方案在 `dag_longest_path()` 函数中实现，该函数返回节点 ID 列表。

要重建该对话，只需迭代这些 ID，并打印出相应的数据。

以上代码是用第 2 章中创建的 stream__RWC2015__RWCFinal_Rugby.jsonl 文件运行的，生成的输出结果如下所示（为简洁起见，省略了最长对话的输出）。

```
Name:
Type: DiGraph
Number of nodes: 198004
Number of edges: 1440
Average in degree:   0.0073
Average out degree:  0.0073
Most replied tweet (15 replies):
{'author': 'AllBlacks', 'tweet': 'Get ready for live scoring here, starting
shortly. You ready #AllBlacks fans? #NZLvAUS #TeamAllBlacks #RWC2015',
'created_at': 'Sat Oct 31 15:49:22 +0000 2015'}
Longest discussion:
# ...
```

正如你看到的，推文的数量远大于边的数量。这是因为很多推文并未与其他推文连接，也就是说，只有一小部分推文是对其他推文的回复。当然，这个结果取决于不同的数据，以及是否考虑包含自我回复和对以往推文（即不在我们数据集中的推文）的回复。

3.4 在地图上绘制推文

本节将介绍如何用地图可视化推文。数据可视化是提供数据直观概览的一种良好方式，为你揭示了数据集的特定特征。

在一小部分推文中，我们可以发现用户设备的地理位置坐标。虽然很多用户在设备上禁用了该功能，但这个数据的挖掘对于理解推文的地理分布非常有用。

本节将介绍 GeoJSON，它是用于地理数据结构的一种常见数据格式，常用于构建推文的交互式地图。

3.4.1 将推文转换为 GeoJSON

GeoJSON 是基于 JSON 的格式，用于对地理数据结构进行编码。GeoJSON 对象可以表示几何结构、特征或特征的集合。几何结构只包含有关形状的信息，如点、线、多边形和更复杂的形状。特征扩展了几何结构的概念，包含几何结构以及一些额外的（自定义）属性。最后，特征的集合就是特征的列表。

GeoJSON 数据结构总是 JSON 对象。以下片段是一个 GeoJSON 的示例，表示两个不同点的集合，每个点表示一个特定的城市。

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -0.12
          51.5
        ]
      },
      "properties": {
        "name": "London"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -74,
          40.71
      ]
    },
    "properties": {
      "name": "New York City"
     }
    }
  ]
}
```

以上这个 GeoJSON 对象的第一个键表示对象的 `type`。这个字段是必需的，且其值必须是以下属性之一。

- ❑ `Point`：用于表示一个位置
- ❑ `MultiPoint`：用于表示多个位置
- ❑ `LineString`：表示串联两个或多个位置的连接线
- ❑ `MultiLineString`：相当于多条连接线
- ❑ `Polygon`：表示封闭的连接线，即第一个和最后一个位置相同
- ❑ `GeometryCollection`：表示不同几何形状的列表
- ❑ `Feature`：表示前面的某项（不包括 `GeometryCollection`）以及一些额外的自定义属性
- ❑ `FeatureCollection`：表示特征列表

前面示例中的 `type` 具有 `FeatureCollection` 值，`features` 字段将是一个对象列表（每一个元素都是一个 `Feature`）。

示例中展示的两个特征都是简单的点，因此，这两个点的 `coordinates` 字段是两个元素的

数组：经度和纬度。这个字段还可以包含第三个元素：海拔（如果省略，则表示海拔为 0）。

清楚需要的结构后，就可以从推文数据集中抽取出地理信息。以下的脚本 `twitter_make_geojson.py` 读取了 JSON Lines 格式的推文数据集，并生成了一个 GeoJSON 文件，该文件包含所有的推文和相关的地理信息。

```
# Chap02-03/twitter_make_geojson.py
import json
from argparse import ArgumentParser

def get_parser():
  parser = ArgumentParser()
  parser.add_argument(`--tweets`)
  parser.add_argument(`--geojson`)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  # 读入推文集合并构建地理数据结构
  with open(args.tweets, 'r') as f:
    geo_data = {
      "type": "FeatureCollection",
      "features": []
    }
    for line in f:
      tweet = json.loads(line)
      try:
        if tweet['coordinates']:
          geo_json_feature = {
            "type": "Feature",
            "geometry": {
              "type": "Point",
              "coordinates": tweet['coordinates']['coordinates']
            },
            "properties": {
              "text": tweet['text'],
              "created_at": tweet['created_at']
            }
          }
          geo_data['features'].append(geo_json_feature)
      except KeyError:
        # 如果 json 文档不是推文，则跳过
        continue
    # 保存地理数据
    with open(args.geojson, 'w') as fout:
      fout.write(json.dumps(geo_data, indent=4))
```

以上脚本用 `ArgumentParser` 读取命令行参数。可以用以下命令运行以上脚本。

```
$ python twitter_make_geojson.py \
  --tweets stream__RWC2015__RWCFinal_Rugby.jsonl \
  --geojson rwc2015_final.geo.json
```

在这个示例中，我们将用`--tweets`参数读取在第 2 章中获取的 stream__RWC2015__RWCFinal_Rugby.jsonl 文件，输出结果将传递给`--geojson`参数并存储在 rwc2015_final.geo.json 文件中。

地理数据结构是`FeatureCollection`，在代码中由`geo_data`字典表示。当循环推文数据集文件时，该代码将导入每条推文，并将其作为一个特征附加到特征集合中。对于每个特征，我们将保存几何信息和相关的坐标，以及一些额外的属性，如推文的文本和创建时间。如果数据集中的任何 JSON 文档不是一条合法的推文（如 Twitter API 返回的错误信息），由于不合法的推文信息不包含期待的属性，程序中的`try/except`会捕获该错误并触发`KeyError`。

代码的最后一部分将地理信息保存到给定的 JSON 文件，用于后续的自定义地图。

3.4.2 用 Folium 轻松绘制地图

本节将介绍 Folium，它是一个 Python 库，可以帮助我们轻松生成交互式地图。

Folium 连接了 Python 数据处理能力和 JavaScript UI。具体来说，Folium 允许 Python 开发者将 GeoJSON 和 TopoJSON 数据与 Leaflet 库整合，Leaflet 是一个特征非常丰富、用于构建交互式地图的前端库。

使用 Folium 这样的库的好处是，它可以处理 Python 数据结构与 JavaScript、HTML 和 CSS 组件的无缝转换。Python 开发者可以不具备任何前端知识，只需通过这个库将结果输出为 HTML 文件（或者直接通过 Jupyter Notebook 展示）。

可以用`pip`在虚拟环境中安装这个库，如下所示。

```
$ pip install folium
```



本节示例使用的 Folium 库是 0.2 版本的，使用时需要仔细阅读文档，因为作为一个新的项目，Folium 的接口可能会有一些重大改变。

以下示例展示了以欧洲为中心的一个简单地图，该地图有两个标记，一个在伦敦，一个在巴黎。

```
# Chap02-03/twitter_map_example.py
from argparse import ArgumentParser
import folium

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--map')
  return parser
```

```
def make_map(map_file):
  # 自定义地图
  sample_map = folium.Map(location=[50, 5],
                          zoom_start=5)
  # 标记伦敦
  london_marker = folium.Marker([51.5, -0.12],
                                popup='London')
  london_marker.add_to(sample_map)
  # 标记巴黎
  paris_marker = folium.Marker([48.85, 2.35],
                               popup='Paris')
  paris_marker.add_to(sample_map)
  # 保存到 HTML 文件
  sample_map.save(map_file)

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  make_map(args.map)
```

以上代码用 `ArgumentParser` 解析命令行参数并选择输出文件。可以用以下命令运行上述代码。

```
$ python twitter_map_example.py --map example_map.html
```

执行代码后，example_map.html 文件会包含输出，该输出结果可以在浏览器中进行可视化。图 3-4 展示了该代码的输出。



图 3-4　用 Folium 构建的示例地图

这段代码的核心逻辑是由 make_map()函数实现的。我们将先创建一个 folium.Map 对象，它在特定位置（即[latitude, longitude]的数组）坐标的中心，并带有特定的缩放比例。zoom_start 属性是一个整数，较小的数表示缩小，这样可以查看完整图片，而较大的数值表示放大。

生成地图后，可以附加一些自定义的标记。示例代码展示了如何在特定位置创建气球形状的标记。在图 3-4 中，巴黎的标记被点击并展示了弹出信息。

了解如何使用 Folium 后，可以将其应用于我们的推文数据集。以下示例展示了如何导入 GeoJSON 文件中的标记列表。


```
# Chap02-03/twitter_map_basic.py
from argparse import ArgumentParser
import folium

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--geojson')
  parser.add_argument('--map')
  return parser

def make_map(geojson_file, map_file):
  tweet_map = folium.Map(location=[50, 5],
                         zoom_start=5)
  geojson_layer = folium.GeoJson(open(geojson_file),
                                 name='geojson')
  geojson_layer.add_to(tweet_map)
  tweet_map.save(map_file)

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  make_map(args.geojson, args.map)
```

上述脚本也用到了 ArgumentParser。可以用以下命令运行上述脚本。

```
$ python twitter_map_basic.py \
  --geojson rwc2015_final.geo.json \
  --map rwc2015_final_tweets.html
```

--geojson 参数用于传递前面创建的文件，它包含 GeoJSON 信息。--map 参数用于提供输出文件的名称，这样我们就可以观察 rwc2015_final_tweets.html 的 HTML 页面，如图 3-5 所示。

这段代码与前面代码的不同之处是，实现 make_map()函数的方法不同。我们从地图对象的初始化开始。我们不会一个一个地添加标记，而是用 GeoJSON 文件生成一个层，然后将该层添加到地图上面。folium.GeoJson 对象负责转换 JSON 数据，这样就可以很轻松地生成地图。

图 3-5 推文的基本地图

和前面一样，图 3-5 中的地图以欧洲为中心。我们可以观察到，英国的标记最拥挤（事件发生在伦敦），但在最小缩放比例下，很难辨识推文的实际地理位置分布。

解决这个问题的一种方式是放大地图，因为这个地图是交互式的。Folium 和 Leaflet 还提供了另一种方式来对密集区域的标记进行分组。

以下脚本用 `MarkerCluster` 对象来解决以上问题。

```
# Chap02-03/twitter_map_clustered.py
from argparse import ArgumentParser
import folium

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--geojson')
  parser.add_argument('--map')
  return parser

def make_map(geojson_file, map_file):
  tweet_map = folium.Map(location=[50, 5],
                         zoom_start=5)
  marker_cluster = folium.MarkerCluster().add_to(tweet_map)

  geojson_layer = folium.GeoJson(open(geojson_file),
                                 name='geojson')
  geojson_layer.add_to(marker_cluster)
  tweet_map.save(map_file)
```

```
if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  make_map(args.geojson, args.map)
```

上述脚本也使用了 `ArgumentParser`，可以用以下命令运行上述脚本。

```
$ python twitter_map_clustered.py \
  --geojson rwc2015_final.geo.json \
  --map rwc2015_final_tweets_clustered.html
```

以上参数的含义和前一段脚本中的一致。这里输出结果保存在 rwc2015_final_tweets_clustered.html 文件中，可以用浏览器打开。

图 3-6 展示了放大后突出伦敦地区的部分地图。可以看到，一些标记被组合在一起作为一个簇，其表示为一个圆圈对象，该对象显示了其中包含的元素数量。



图 3-6 带有类标记的 Folium 地图

当鼠标移到某个簇上时，地图将高亮显示该簇表示的区域，因此用户可以在不放大的情况下了解该地区的密度。在图 3-6 中，我们高亮显示了伦敦西南部（该地点是 Twickenham 体育馆，正好是事件的发生地）一个包含 65 项的簇。

还可以组合使用 Folium 的不同特征，以提供更丰富的用户体验。例如，可以结合 GeoJSON、簇和弹出窗口，从而允许用户点击特定标记并查看对应的推文。

用类和弹出窗口创建地图的示例如下所示。

```
def make_map(geojson_file, map_file):
  tweet_map = folium.Map(location=[50, 5],
                         zoom_start=5)
  marker_cluster = folium.MarkerCluster().add_to(tweet_map)
  geodata = json.load(open(geojson_file))
  for tweet in geodata['features']:
    tweet['geometry']['coordinates'].reverse()
    marker = folium.Marker(tweet['geometry']['coordinates'],
                           popup=tweet['properties']['text'])
    marker.add_to(marker_cluster)
  tweet_map.save(map_file)
```

前面定义过的 `make_map()` 函数可以直接替换原来程序中的 `make_map()`，因为接口相同。

注意，这里创建的 `Marker` 对象需要 `[latitude, longitude]` 形式的坐标，而 GeoJSON 格式使用的是 `[longitude, latitude]`。因此，定义 `marker` 前需要调换坐标数组的顺序。

图 3-7 展示了一个地图示例，图中放大并高亮显示了体育馆。有一个标记被点击了，因此弹出窗口中显示了相应的推文。

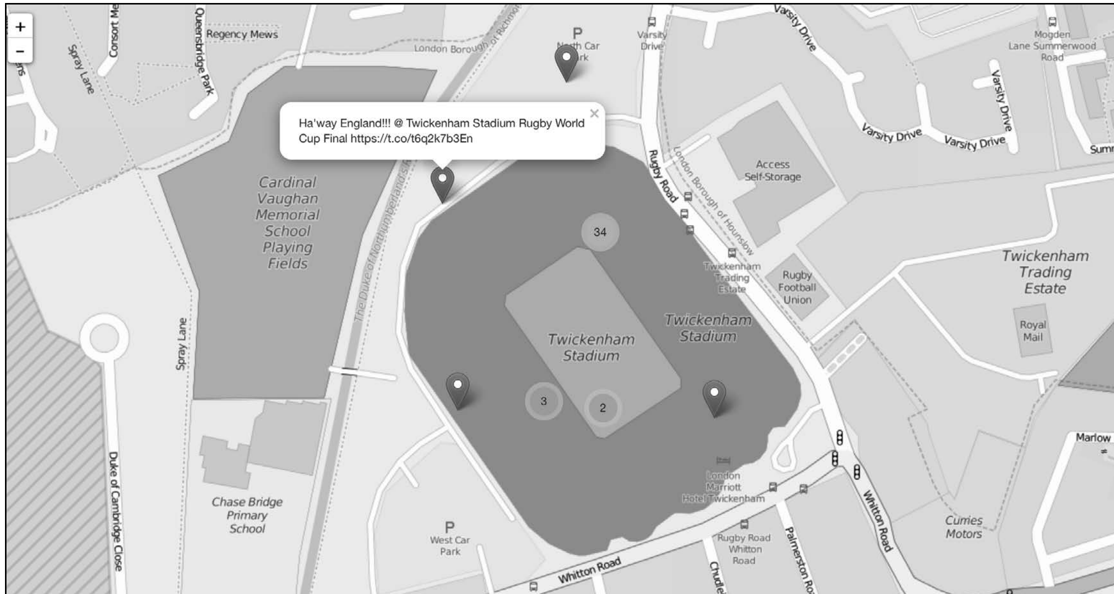


图 3-7　前面的地图（图 3-6）放大后的效果，簇可以作为单个标记

当使用大量标记时，需要考虑可能会在 UI 端出现一些性能问题。具体来说，当结合标记和弹出窗口时，加载几百项后，地图的导航会变得特别慢，而且浏览器会拒绝加载。如果发生这种情况，建议尝试较小的数据集并减少地图中的特征数量。

3.5　小结

本章继续探讨了 Twitter 数据的挖掘。第 2 章重点关注文本和频率，而本章重点介绍了如何分析用户连接和互动。我们介绍了如何抽取显式连接（即粉丝和好友）的信息，以及如何比较用户的影响力和参与度。

通过讨论用户社区，我们介绍了利用聚类算法根据用户资料的描述对用户分组的无监督学习方法。

我们对有关实时事件的数据应用了网络分析技术，目的是挖掘推文流中的对话，理解如何识别拥有最多回复的推文，以及如何确定最长的对话。

最后展示了如何通过将推文绘制在地图上来理解推文的地理分布。我们用 Python 库 Folium 展示了漂亮地可视化地理数据的可行性以及简便性。

下一章将关注 Facebook，它可能是当今最流行的社交网络平台。

第 4 章

Facebook 帖子、页面和用户互动

在介绍社会媒体挖掘的书中，你可能会期待有关 Facebook 挖掘的介绍。Facebook 于 2004 年推出，最初只用于哈佛的学生圈，如今已是一个市值 10 多亿美元的公司，拥有近 15 亿月活跃用户。Facebook 的受欢迎程度使得它成为了非常有趣的数据挖掘场所。

本章包含如下主题：

- 创建一个与 Facebook 平台交互的应用
- 与 Facebook Graph API 交互
- 挖掘鉴权用户的帖子
- 挖掘 Facebook 的页面，可视化帖子，并计算参与度
- 根据一组帖子构建一个词云

4.1 Facebook Graph API

Facebook Graph API 是 Facebook 平台的核心，也是允许 Facebook 与第三方资源整合的主要组件。顾名思义，它提供了数据的图结构视图，表示对象及对象间的关系。不同的平台组件允许开发者访问 Facebook 数据，并将 Facebook 的功能整合到第三方应用中。

2014 年，该 API 的 2.0 版本发布，使数据挖掘的机会发生了很大转变。数据分析最主要的关注对象是社交图谱，即用户间的关系。在 Graph API 2.0 之后，希望获取这种连接关系的应用必须申请 `user_friends` 许可，但 API 只会返回同样使用该应用的好友列表。

这实际上改变了数据分析中曾经的“金矿”。本节将介绍如何用 Python 创建 Facebook 应用，以及如何与 Facebook Graph API 进行基本的交互。

4.1.1 注册你的应用

Facebook API 的接入需要通过一个注册应用。开发者必须先注册他们的应用，才能获得 Graph

API 的接入凭证。作为 Facebook 用户，你必须注册成开发者，然后才能创建应用，而且你的账户必须通过电话号码或者信用卡认证。

在 Facebook 开发者网站注册的过程很简单：点击 My Apps 菜单下的 Add a New App 链接。点击后会打开对话框，如图 4-1 所示。这里将 Social Media Mining 作为示例应用的显示名称（限制为最多 32 个字符）。

Create a New App ID
Get started integrating Facebook into your app or website
Display Name
Social Media Mining
Namespace
A unique identifier for your app (optional)
No
Is this a test version of another app? Learn More.
Category
Choose a category
By proceeding, you agree to the Facebook Platform Policies and the Facebook Privacy Policy
Cancel
Create App ID

图 4-1　创建新 Facebook 应用的对话窗口

设置好应用的名称和类别后，点击 Create App ID 确认创建该应用。

至此，可以在 My Apps 菜单下看到我们选择的应用名称，点击该名称就会打开应用仪表盘，如图 4-2 所示。



图 4-2　应用仪表盘的视图

该面板提供了一些关键信息，如App ID和App Secret，这些都是接入API必须用到的。为了看到App Secret，你需要提供Facebook密码来确认身份。无须强调的是，不应该和其他任何人共享这些细节。

App ID和App Secret之间还有一个API Version（图4-2中使用的是v2.5版本）。默认情况下，新应用的API版本是最新的可用版本。在2014年的F8大会上，Facebook宣布将在未来两年支持特定的API版本。这个信息非常值得注意，因为当特定版本的API不再受到支持时，如果你的应用不更新就不能正确运行。



Facebook平台的版本

官方文档（https://developers.facebook.com/docs/apps/versions）中详细介绍了Facebook平台的版本策略，以及特定版本的更新情况（https://developers.facebook.com/docs/graph-api/changelog）。

开始时，你的应用是在开发模式中创建的。只有你和Roles菜单中指定的其他Developers或Testers可以接入该应用。在创建应用的仪表盘多花一些时间理解基本的配置选项及其隐含含义是非常有价值的。

4.1.2　鉴权和安全

为了获得用户的资料信息，以及其与其他对象（如页面、地点等）的互动信息，你的应用必须获得带有特定权限的访问令牌。

一个令牌对用户–应用的组合来说是唯一的，它可以处理用户授权应用的许可。生成一个访问令牌需要用户交互，也就是说，用户必须在Facebook确认授权请求许可的应用（通常通过对话窗口）。

出于测试的目的，获取访问令牌的另一种方法是使用Graph API Explorer。它是Facebook开发的一个工具，可以为开发者提供与Graph API交互的便捷接口。图4-3展示了Graph API Explorer的界面，从可用的应用列表中选择我们的应用后，可以在Get Token菜单点击Get User Access Token。

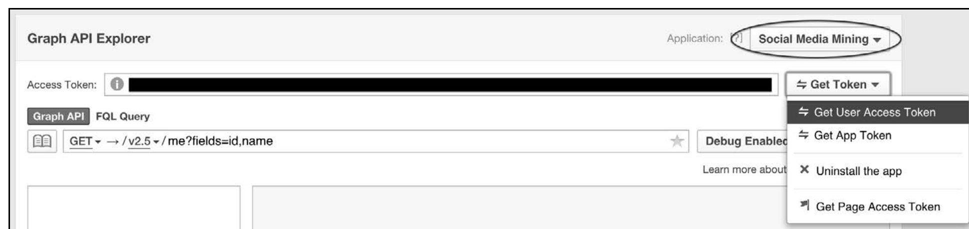


图4-3　在Graph API Explorer生成一个访问令牌

点击后会打开一个对话窗口（如图 4-4 所示），可用于指定访问令牌包含的权限。确认应用的访问权限后会生成一串字母字符串，即访问令牌，它在创建后的 2 小时内有效。点击令牌旁的小图标后会出现以下信息。

Select Permissions
User Data Permissions Extended Permissions
user_about_me user_actions.books user_actions.fitness
user_actions.music user_actions.news user_actions.video
user_birthday user_education_history user_events
user_friends user_games_activity user_hometown
user_likes user_location user_managed_groups
user_photos user_posts user_relationship_details
user_relationships user_religion_politics user_status
user_tagged_places user_videos user_website
user_work_history
Public profile included by default. Get Access Token Clear Cancel

图 4-4 在 Graph API Explorer 为访问令牌选择权限

从图 4-4 可以看到，对 Facebook 应用权限的划分是非常细粒度的。这样一来，安装应用的用户很清楚他们的哪些数据是和你的应用共享的。

对以下示例来说，我们将使用一些字段，如用户全名、地点和帖子。对于可以准确检索这些信息的代码，首先需要解决权限问题（如 `user_location`、`user_posts` 等）。



当用户首次访问一个应用时，Facebook 将弹出对话窗口以展示应用的权限列表。该用户可以看到应用请求的所有权限。

4.1.3 用 Python 连接 Facebook Graph API

定义好应用的细节后，可以通过 Python 接入 Facebook Graph API。

Facebook 没有提供官方的 Python 客户端。用 requests 库实现自己的客户端是一种有趣的实践，可以帮助我们了解 API 的细节，不过已经有一些选项可以简化该过程。

对接下来的示例来说，我们将使用基于 requests 库的 facebook-sdk，requests 库提供简单易用的与 Web 服务交互的数据接口。在撰写本书时，该库在 PyPI 上的最新版本是 1.0.0，并且支持 Python 3。其前一个版本对 Python 3 的支持有些问题。可以用 `pip` 在虚拟环境中安装这个库。

```
$ pip install facebook-sdk
```

遵循前面介绍的步骤获取临时的令牌后，可以立即测试这个库。

首先，与第 2~3 章中介绍的配置 Twitter 的访问类似，我们将令牌保存在代码读取的环境变量中。在命令行窗口输入以下命令。

```
$ export FACEBOOK_TEMP_TOKEN="your-token"
```

以下脚本 facebook_my_profile.py 连接到 Graph API，并查询鉴权用户的资料。

```
# Chap04/facebook_my_profile.py
import os
import json
import facebook

if __name__ == '__main__':
  token = os.environ.get('FACEBOOK_TEMP_TOKEN')

  graph = facebook.GraphAPI(token)
  profile = graph.get_object("me", fields='name,location{general_info,location},
    languages{name,description}')

  print(json.dumps(profile, indent=4))
```

这个脚本不需要任何参数，可以用以下命令运行。

```
$ python facebook_my_profile.py
```

输出结果是 API 返回的 JSON 对象。

```
{
  "name": "Marco Bonzanini",
  "location": {
    "name": "London, United Kingdom",
    "id": "106078429431815"
  },
  "id": "10207505820417553"
}
```

get_object() 函数将 Facebook Graph 中的特定对象的 ID 或名称作为第一个参数，并返回相应的信息。在我们的示例中，ID me 只是鉴权用户的别名。如果不指定第二个参数和字段，那么 API 只会返回对象的 ID 和名称。在这个示例中，我们明确要求输出包含 name 和 location。正如你看到的，location 并不是一个字符串，而是拥有自己字段的复杂对象（因为这里没有特别指定，所以地点包含的字段只有 id 和 name）。

从 GraphAPI 类获取数据的接口非常直接。这个类还提供了在 Facebook 上发布和更新数据的功能，允许应用与 Facebook 平台交互（例如，在鉴权用户的墙板上发布一些内容）。

示例中的主要方法如下所示。

- get_object(id, **args)：用于检索给定 id 的对象，而且接收可选关键字参数
- get_objects(ids, **args)：用于检索给定 ids 列表的对象，而且接收可选关键字参数

- get_connections(id, connection_name, **args)：用于检索给定 id 对象的 connection_name 关系中包含的对象列表，而且接收可选关键字参数
- request(path, args=None, post_args=None, files=None, method=None)：该通用方法用 API 文档中定义的 path 实现对 API 的特定请求，可选参数定义了如何执行 API 调用

facebook_my_profile.py 脚本中的示例用 get_object() 方法下载当前用户的资料。在这个示例中，可选关键字参数 fields 用于指定我们希望从 API 检索的属性。用户资料的完整属性列表可以参见文档（https://developers.facebook.com/docs/graph-api/reference/v2.5/user）。

根据 API 规范，我们可以看到如何定制字段的字符串以获取更多信息。也可以执行嵌套请求，并获取与给定用户连接的对象的信息。我们的示例检索了位置，即**页面**类型的一个对象。由于每个页面都有一些相应的属性，也可以在请求中包含这些属性。例如，更改 get_object() 请求。

```
profile = graph.get_object("me", fields='name,location{location}')
```


first_level{second_level} 语法可以查询嵌套对象。在这个特例中，location 的命名可能会造成混淆，因为它既是一级属性也是二级属性。解决方法是理解数据类型。第一级的 location 是用户资料的一个属性，其数据类型是一个 Facebook 页面（拥有 ID、名称和其他属性）。第二级的 location 是前面提到的 Facebook 页面的一个属性，而且它是实际位置的一个描述符，由 latitude 和 longitude 组成。对于带有二级 location 信息的前述代码，输出如下所示。

```
{
  "name": "Marco Bonzanini",
  "location": {
    "name": "London, United Kingdom",
    "id": "106078429431815"
  },
  "id": "10207505820417553",
  "location": {
    "id": "106078429431815",
    "location": {
      "city": "London",
      "latitude": 51.516434161634,
      "longitude": -0.12961888255995,
      "country": "United Kingdom"
    }
  }
}
```



为 location 对象默认检索到的属性是 city、country、latitude 和 longitude。

正如本节开头提到的，随着新版 Facebook Graph API 的出现，一些数据挖掘的机会受到了限制。尤其是挖掘社交图谱（即好友关系）只适用于我们应用的用户。以下脚本尝试为鉴权用户获取好友列表。

```
# Chap04/facebook_get_friends.py
import os
import facebook
import json

if __name__ == '__main__':
  token = os.environ.get('FACEBOOK_TEMP_TOKEN')

  graph = facebook.GraphAPI(token)
  user = graph.get_object("me")
  friends = graph.get_connections(user["id"], "friends")
  print(json.dumps(friends, indent=4))
```

虽然这里 API 的调用需要我们的应用具有获取 `user_friends` 的权限，但该脚本仍然不能获取有关好友的大量数据，因为鉴权用户（即 `me`）目前是应用的唯一用户。以下是一个示例输出。

```
{
  "data": [],
  "summary": {
    "total_count": 266
  }
}
```

可以看到，我们能检索的唯一信息是给定用户的好友数量，而好友的数据是由空列表表示的。如果一些好友也是我们应用的用户，那么就可以通过这个调用获取他们的资料。

最后，介绍一下 Graph API 的访问频率限制。正如文档（https://developers.facebook.com/docs/graph-api/advanced/rate-limiting）中介绍的那样，很少会达到访问上限。该上限是根据每个应用和用户计算的，如果应用达到了每日访问次数上限，那么该应用的所有调用都会被限制，而不只是给定用户的调用。每日允许的次数基于前一天的用户数量和当天登录数量计算，其总和构成了用户基数。应用的每个用户可以在 60 分钟的时间窗口内进行 200 次的 API 调用。这个限制对我们的示例来说足够了。建议你查看官方文档，了解应用的访问频率限制可能带来的影响。

下一节将下载鉴权用户的所有帖子，并对这部分数据进行分析。

4.2　挖掘你的帖子

前面用一个简单示例介绍了 Python facebook-sdk，接下来将介绍数据挖掘。第一个练习是下载自己的帖子（即鉴权用户发布的帖子）。

`facebook_get_my_posts.py` 脚本接入 Graph API，并获取鉴权用户 `me` 发布的帖子列表。用第 2~3 章中介绍过的 JSON Lines 格式将这些帖子保存在 my_posts.jsonl 文件中，文件的每一行是一个 JSON 文档。

```
# Chap04/facebook_get_my_posts.py
import os
import json
import facebook
import requests

if __name__ == '__main__':
  token = os.environ.get('FACEBOOK_TEMP_TOKEN')

  graph = facebook.GraphAPI(token)
  posts = graph.get_connections('me', 'posts', fields='message,created_time,
    description,caption,link,place,status_type,shares')

  while True: # 不停地翻页
    try:
      with open('my_posts.jsonl', 'a') as f:
        for post in posts['data']:
          f.write(json.dumps(post)+"\n")
          # 进入下一页
          posts = requests.get(posts['paging']['next']).json()
    except KeyError:
      # 没有下一页就跳出循环
      break
```


这个脚本不需要任何命令行参数，因此可以用以下命令运行。

```
$ python facebook_get_my_posts.py
```

该脚本提供了翻页的一个有趣示例。因为帖子列表太长，无法通过单个 API 调用收集，所以 Facebook 提供了翻页信息。

用 get_connections()方法执行的初始 API 调用返回第一页的帖子（保存在 posts ['data']中），也返回了迭代不同页面需要的信息，保存在 posts['paging']中。因为 Python facebook-sdk 库中并没有实现翻页功能，所以需要直接使用 requests 库。好在 Graph API 提供的响应包含了我们需要请求的 URL，这样就可以获得下一个页面的帖子。实际上，如果审查 posts['paging']['next'] 变量的值，可以看到待查询的准确 URL 字符串，其中包括访问令牌、API 版本号和所需的其他详细信息。

翻页是在 while True 循环中实现的，达到最后的页面时会被 KeyError 异常中断。因为最后一个页面不会包含 posts['paging']['next']的引用，所以如果试图获取字典的这个键，就会抛出异常并跳出循环。

执行代码后，可以查看 my_posts.jsonl 文件的内容。这个文件的每一行都是一个 JSON 文档，包含唯一的 ID、该帖子的文本消息内容和 ISO 8601 格式的创建时间。以下的 JSON 文档表示其中一个下载的帖子。

```
{
  "created_time": "2015-11-04T08:01:21+0000",
  "id": "10207505820417553_10207338487234328",
  "message": "The slides of my lighting talk at the PyData London
    meetup last night\n"
}
```

与 get_object()函数类似，get_connections()函数接收 fields 参数，以检索目标对象的更多属性。以下脚本重构了前面示例中的代码，以获取帖子的更多属性。

```
# Chap04/facebook_get_my_posts_more_fields.py
import os
import json
import facebook
import requests

if __name__ == '__main__':
  token = os.environ.get('FACEBOOK_TEMP_TOKEN')

  graph = facebook.GraphAPI(token)
  all_fields = [
    'message',
    'created_time',
    'description',
    'caption',
    'link',
    'place',
    'status_type',
    'message_tags',
    'picture',
    'privacy',
    'properties',
    'story_tags',
    'from',
    'to',
    'with_tags'
  ]
  all_fields = ','.join(all_fields)
  posts = graph.get_connections('me', 'posts', fields=all_fields)

  while True:  # 继续翻页
    try:
      with open('my_posts.jsonl', 'a') as f:
        for post in posts['data']:
          f.write(json.dumps(post)+"\n")
        # 跳转到下一页面
        posts = requests.get(posts['paging']['next']).json()
    except KeyError:
      # 没有新的页面则跳出循环
      break
```

我们想要检索的所有字段是在 all_fields 列表中声明的，然后按照 Graph API 的要求将这些属性名称用逗号连接成一个字符串。接着通过 fields 关键字参数将这个值传递给 get_connections()方法。

下一节将更详细地介绍帖子的结构。

4.2.1　帖子的结构

帖子是复杂的对象，因为它可以是用户发布的任意一段内容。表 4-1 总结了帖子的有趣属性及其含义。

表 4-1　帖子的属性及其含义

属性名称	描　述
id	表示唯一标识符的字符串
application	App 对象，其中包含用于发布帖子的应用的信息
status_type	表示帖子类型的字符串（如 added_photos 或 shared_story）
message	表示帖子状态消息的字符串
created_time	表示帖子发布时间的字符串，ISO 8601 格式
updated_time	表示帖子最新修改时间的字符串，ISO 8601 格式
message_tags	消息中标注的资料列表
from	发布消息的资料
to	帖子中提及或作为目标的资料列表
place	帖子带有的位置信息
privacy	带有帖子隐私设置的对象
story_tags	同 message_tags
with_tags	被标记为帖子作者的资料列表
properties	所有附带视频的属性列表（如视频的长度）

一个 Post 对象所拥有的属性比表 4-1 列出的要多。可以查看官方文档（https://developers.facebook.com/docs/graph-api/reference/v2.5/post）获取完整的属性列表，这种帖子对象的复杂性在文档中更清晰。

4.2.2　时间频率分析

下载所有帖子后，我们将基于不同帖子的创建时间进行分析。这个分析的目的是突出用户的行为，如用户何时在 Facebook 上发布了最多的帖子。

facebook_post_time_stats.py 脚本用 ArgumentParser 获取命令行输入，即带有帖子的.jsonl 文件。

```
# Chap04/facebook_post_time_stats.py
import json
from argparse import ArgumentParser
import dateutil.parser
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from datetime import datetime

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--file',
                      '-f',
                      required=True,
                      help='The .jsonl file with all the posts')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  with open(args.file) as f:
    posts = []
    for line in f:
      post = json.loads(line)
      created_time = dateutil.parser.parse(post['created_time'])
      posts.append(created_time.strftime('%H:%M:%S'))
    ones = np.ones(len(posts))
    idx = pd.DatetimeIndex(posts)
    # 实际序列（目前是单位1组成的序列）
    my_series = pd.Series(ones, index=idx)

    #重抽样为1小时间隔的区间
    per_hour = my_series.resample('1H', how='sum').fillna(0)
    # 画图
    fig, ax = plt.subplots()
    ax.grid(True)
    ax.set_title("Post Frequencies")
    width = 0.8
    ind = np.arange(len(per_hour))
    plt.bar(ind, per_hour)
    tick_pos = ind + width / 2
    labels = []
    for i in range(24):
      d = datetime.now().replace(hour=i, minute=0)
      labels.append(d.strftime('%H:%M'))
    plt.xticks(tick_pos, labels, rotation=90)
    plt.savefig('posts_per_hour.png')
```

可以用以下命令运行该脚本。

```
$ python facebook_post_time_stats.py -f my_posts.jsonl
```

该脚本首先生成带有每个帖子创建时间的帖子列表。`dateutil.parser.parse()`函数可以读取 ISO 8601 日期字符串，并存放到 `datetime` 对象中，然后用 `strftime()`函数将其转换为 HH:MM:SS 字符串。

然后用创建时间列表索引 pandas Series，该序列的初始值是一个由 1 组成的序列。接着按小时对该序列进行重抽样，并汇总帖子的数量。至此，我们得到了一个有 24 项的序列，每一项表

示一天中的一个小时，并且带有那个小时发布的帖子数量。最后一部分代码的目的是，用简单的柱状图画出序列，以便对一天中的帖子数量分布进行可视化表示。

图 4-5 展示了画图结果。

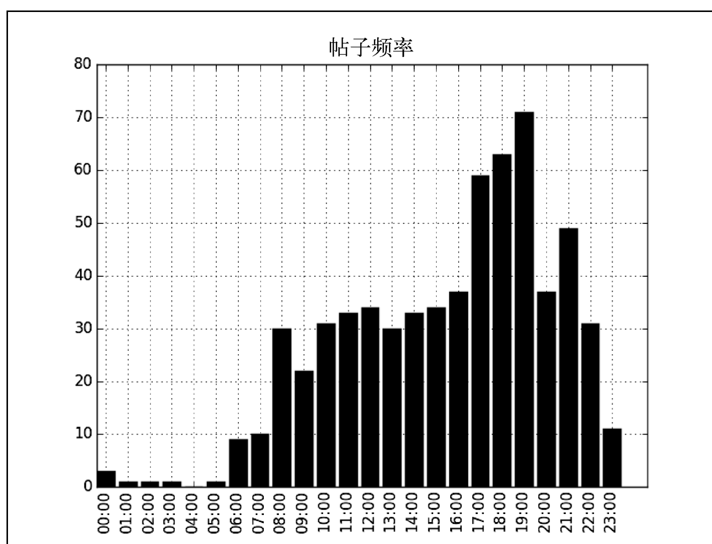


图 4-5 帖子的时间频率

可以看到，发布时间在傍晚和晚上达到了峰值，但其扩展度是全天持续的（最低频率在深夜和凌晨）。一方面需要注意这里并没有考虑地点，即创建时间被归一化为 UTC（协调世界时间），因此没有考虑原始的时区。例如，如果一个帖子在美国东海岸的下午 4 点发布，那么该帖子的创建时间会被 UTC 记录为晚上 9 点。这是因为 EST（美国东部标准时间）相当于 UTC – 05:00，即滞后于 UTC 5 个小时（在不执行夏令时的秋冬季）。

4.3 挖掘 Facebook 页面

Facebook 不只用于个人和亲朋好友联系，很多公司、品牌和机构还用其与人们互动。Facebook 页面由一个 Facebook 个人账户创建和管理，用途很多：

- 分享商业信息（如一家餐馆或者在线商店）
- 表示一个名人（如一个足球明星或者摇滚乐队）
- 与一组受众联系（如作为在线社区的扩展）

与个人账户不同的是，页面上发布的帖子是公开可见的。普通用户可以喜欢一个页面，这意味着他们将通过新闻订阅收到发布内容的更新，这里的新闻订阅就是他们自定义的 Facebook 主页。

例如，Packt 出版社的 Facebook 主页是 https://www.facebook.com/PacktPub，其中包含了关于 PacktPub 的基本信息，以及由 PacktPub 发布的与纸质书、电子书和视频教程相关的帖子。

我们可以从 API 文档看到给定页面的很多信息，特别是关于页面的以下属性。

- id：这个页面的数字标识符
- name：在 Facebook 显示的页面的名称
- about：页面的文本描述
- link：链接到该页面的 Facebook 链接
- website：如果存在的话，即为机构的网页
- general_info：有关页面的常见信息的文本字段
- likes：喜欢该页面的用户数量

一个 Page 对象也可以与其他对象连接，文档描述了与其他对象连接的所有情况，如下所示。

- posts：页面发布的帖子列表
- photos：页面的图片
- albums：页面发布的图片专辑列表
- picture：该页面的信息图片

特定的页面类型可以显示额外的信息，这些信息是为其商业用途定制的（例如，一个餐馆可以显示开放时间，一个品牌可以显示品牌成员的列表等）。

以下脚本查询 Facebook Graph API，以获得关于特定 Facebook 页面的基本信息。

```
# Chap04/facebook_get_page_info.py
import os
import json
import facebook
from argparse import ArgumentParser

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  token = os.environ.get('FACEBOOK_TEMP_TOKEN')
  fields = [
    'id',
    'name',
    'about',
    'likes',
```

```
    'website',
    'link'
  ]
  fields = ','.join(fields)

  graph = facebook.GraphAPI(token)
  page = graph.get_object(args.page, fields=fields)

  print(json.dumps(page, indent=4))
```

该脚本用 `ArgumentParser` 的一个实例从命令行获取页面名称（或页面 ID），如下所示。

```
$ python facebook_get_page_info.py --page PacktPub
```

输出结果如下所示。


```
{
  "id": "204603129458",
  "website": "http://www.PacktPub.com",
  "likes": 6357,
  "about": "Packt Publishing provides books, eBooks, video
    tutorials, and articles for IT developers, administrators, and
    users.",
  "name": "Packt Publishing",
  "link": "https://www.facebook.com/PacktPub/"
}
```


还可以用 Facebook Graph API Explorer 获取可用字段的介绍。

4

4.3.1　从页面获取帖子

介绍完如何获取一个页面的基本信息后，我们将介绍如何下载一个页面发布的帖子，并将下载的帖子保存为常见的 JSON 行格式，以便于后续分析帖子。

这个步骤与我们前面下载鉴权用户发布的帖子的过程类似，但其中还包含一些用于计算用户参与度的信息。

```
# Chap04/facebook_get_page_posts.py
import os
import json
from argparse import ArgumentParser
import facebook
import requests

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  parser.add_argument('--n', default=100, type=int)
  return parser

if __name__ == '__main__':
```

```
parser = get_parser()
args = parser.parse_args()

token = os.environ.get('FACEBOOK_TEMP_TOKEN')

graph = facebook.GraphAPI(token)
all_fields = [
  'id',
  'message',
  'created_time',
  'shares',
  'likes.summary(true)',
  'comments.summary(true)'
]
all_fields = ','.join(all_fields)
posts = graph.get_connections('PacktPub',
                              'posts',
                              fields=all_fields)

downloaded = 0
while True:  # 保持翻页
  if downloaded >= args.n:
    break
  try:
    fname = "posts_{}.jsonl".format(args.page)
    with open(fname, 'a') as f:
      for post in posts['data']:
        downloaded += 1
        f.write(json.dumps(post)+"\n")
      # 跳转到下一页
      posts = requests.get(posts['paging']['next']).json()
  except KeyError:
    # 没有更多页面，则跳出循环
    break
```

这个脚本用 `ArgumentParser` 的一个实例从命令行获取页面名称（或页面 ID），以及我们想要下载的帖子的数量。在示例代码中，帖子的数量是可选的（默认为 `100`）。正如前面下载鉴权用户的帖子那样，我们将定义想要在结果中包含的字段列表。特别地，由于要用该数据进行有关用户参与度的分析，我们希望结果中包含有关帖子被喜欢、分享和评论次数的信息。这是通过添加 `shares`、`likes.summary(true)`和 `comments.summary(true)`字段实现的。对于喜欢和评论，`summary(true)`属性用于总结统计值，即聚合计数值。

下载过程在 `while True` 循环中实现，这与下载鉴权用户帖子的方法类似。不同之处是 `downloaded` 计数器，它在每次检索帖子时加 1。它用于限制我们希望检索的帖子的数量，因为页面中往往会发布大量内容。

可以用以下命令运行这段代码。

```
$ python facebook_get_page_posts.py --page PacktPub --n 500
```

执行上述命令将查询 Facebook Graph API，并生成带有 500 个帖子的 posts_PacktPub.jsonl 文件，每行是一条帖子。以下代码展示了一条帖子（即.jsonl 文件的一行）的漂亮打印效果。

```
{
  "id": "post-id",
  "created_time": "date in ISO 8601 format",
  "message": "Text of the message",
  "comments": {
    "data": [ /* 评论列表 */ ],
    "paging": {
      "cursors": {
        "after": "cursor-id",
        "before": "cursor-id"
      }
    },
    "summary": {
      "can_comment": true,
      "order": "ranked",
      "total_count": 4
    }
  },
  "likes": {
    "data": [ /* 用户列表 */ ],
    "paging": {
      "cursors": {
        "after": "cursor-id",
        "before": "cursor-id"
      }
    },
    "summary": {
      "can_like": true,
      "has_liked": false,
      "total_count": 10
    }
  },
  "shares": {
    "count": 9
  }
}
```

4

可以看到，帖子是复杂的对象，具有不同的嵌套信息。度量用户参与度的字段是 `shares`、`likes` 和 `comments`。`shares` 字段表示分享过该故事的用户数量。由于隐私设置，并未包含其他信息。当分享一段内容时，用户实际上也在创建自己的帖子，因此这条新帖子不应该被该网络外的人看到。另一方面，`comments` 和 `likes` 字段是与帖子本身相连的对象，因此它们的信息更详细一些。

对于 `comments` 字段来说，`data` 键包含与评论相关的对象列表，每条评论的结构如下所示。

```
{
  "created_time": "date in ISO 8601 format",
  "from": {
```

```
      "id": "user-id",
      "name": "user-name"
    },
    "id": "comment-id",
    "message": "text of the message"
  }
```

comments 对象还包括一个 paging 字段，为了避免评论的数量超过一个页面，这个字段包含了对指针的引用。给定原始请求，而且请求中包含对 summary(true)属性的引用，那么结果会包含一个简短的摘要数值统计。我们比较关注 total_count。

likes 对象与 comments 对象有些类似，不过前者在这个示例中的数据更简单。明确来说，这里有一个用户 ID 列表，表示喜欢该帖子的用户。类似地，我们也会有 summary(true)属性，表示喜欢数量的摘要数值统计。

下载好数据后就可以做不同的离线分析了。

Facebook Reactions 和 Graph API 2.6

本章的草稿完成不久后，Facebook 推出了名为 Reactions 的新功能。作为**喜欢**按钮的扩展，Reactions 允许用户表达对特定帖子的情绪，而不仅仅是喜欢。用户现在可以表达的新情绪有喜爱、大笑、惊讶、伤心、愤怒（仍有喜欢这一项）。图 4-6 展示了这些新的按钮。

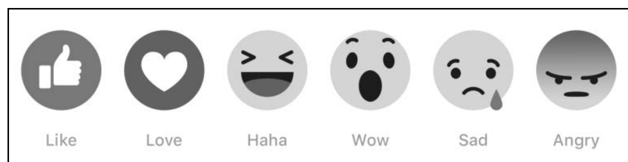


图 4-6　Facebook Reactions 的可视化

支持 Facebook Reactions 的第一个 Graph API 版本是 2.6。本章中的示例大多数基于 2.5 版本的 API。如果计划在数据分析中用到这些特征，你需要确保自己使用的是正确版本。关于如何获取有关 Reactions 的数据，可以查看官方文档（https://developers.facebook.com/docs/graph-api/reference/post/reactions）。

从开发者的角度来看，Reactions 和喜欢非常相似。如果用 API 请求有关 Reactions 的信息，我们得到的每条帖子的结构如下所示。

```
{
  "message": "The content of the post",
  "created_time": "creation date in ISO 8601",
  "id": "the ID of the post",
  "reactions": {
    "data": [
      {
```

```
        "id": "some-user-id",
        "name": "John Doe",
        "type": "WOW"
      },
      {
        "id": "another-user-id",
        "name": "Jane Doe",
        "type": "LIKE"
      },
      /* 更多的Reactions…… */
    ]
  },
  "likes": {
    "data": /* 与喜欢相关的数据，和之前一样 */
  }
}
```

虽然本章的示例主要将喜欢和评论作为理解用户参与的方式，但这种方法也很容易扩展到新的特征，从而更好地帮助发布者理解用户和好友对其内容的看法。

4.3.2 度量参与度

接下来分析 Facebook 页面发布的帖子，并探讨如何度量用户的参与度。Graph API 提供的数据包含以下信息：

- 帖子被分享的次数
- 喜欢该帖子的用户数量
- 与该帖子相关的用户数量
- 该帖子的评论数量

实际上，分享、喜欢和评论都是用户基于其他人发布的一段内容的反应。理解这些动作背后的真实原因是非常困难的（例如，如何区分讽刺和真实的赞赏），深度分析这些反应背后的心理因素超出了本书的范围，因此就本书中的数据挖掘示例来说，我们假设具有更多互动的帖子更“成功”。这里“成功”是带双引号的一个词，因为我们并没有对其进行准确定义，因此本节只计算用户与页面交互的次数。

下面的脚本打印了拥有最多互动次数的 Facebook 页面上的帖子信息。

```
# Chap04/facebook_top_posts.py
import json
from argparse import ArgumentParser

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  return parser
```

```
if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  fname = "posts_{}.jsonl".format(args.page)

  all_posts = []
  with open(fname) as f:
    for line in f:
      post = json.loads(line)
      n_likes = post['likes']['summary']['total_count']
      n_comments = post['comments']['summary']['total_count']
      try:
        n_shares = post['shares']['count']
      except KeyError:
        n_shares = 0
      post['all_interactions'] = n_likes + n_shares + n_comments
      all_posts.append(post)
  most_liked_all = sorted(all_posts,
                          key=lambda x: x['all_interactions'],
                          reverse=True)
  most_liked = most_liked_all[0]
  message = most_liked.get('message', '-empty-')
  created_at = most_liked['created_time']
  n_likes = most_liked['likes']['summary']['total_count']
  n_comments = most_liked['comments']['summary']['total_count']
  print("Post with most interactions:")
  print("Message: {}".format(message))
  print("Creation time: {}".format(created_at))
  print("Likes: {}".format(n_likes))
  print("Comments: {}".format(n_comments))
  try:
    n_shares = most_liked['shares']['count']
    print("Shares: {}".format(n_shares))
  except KeyError:
    pass
  print("Total: {}".format(most_liked['all_interactions']))
```

该脚本用 `ArgumentParser` 从命令行获取参数，可以用以下命令运行。

```
$ python facebook_top_posts.py --page PacktPub
```

当迭代帖子时，我们为每个帖子引入一个 `all_interactions` 键，计算的是喜欢、分享和评论的总和。如果该帖子还未被任何用户分享，则 `shares` 键不会出现在字典中，因此 `post['shares']['count']`接口放在 `try/except` 中，当这个键不存在时，分享次数的值默认为 0。

作为 `sorting()`函数中的一个排序选项，新的 `all_interactions` 键返回的是按照互动次数逆序排列的帖子列表。

脚本的最后一部分打印信息，输出结果如下所示。

```
Post with most interactions:
Message: It's back! Our $5 sale returns!
Creation time: 2015-12-17T11:51:00+0000
Likes: 10
Comments: 4
Shares: 9
Total: 23
```

虽然找到具有最多互动次数的帖子是一个有趣的练习，但这并未揭示整体情况。

下一步是验证一天中的特定时间段是否比其他时间段更成功。也就是说，是否特定时间段发布的帖子获得的互动次数更多。

以下脚本用 `pandas.DataFrame` 来聚合互动的统计值，并画出以 1 小时为区间的结果，具体做法与我们前面分析鉴权用户时类似。

```
# Chap04/facebook_top_posts_plot.py
import json
from argparse import ArgumentParser
import numpy as np
import pandas as pd
import dateutil.parser
import matplotlib.pyplot as plt
from datetime import datetime

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  fname = "posts_{}.jsonl".format(args.page)

  all_posts = []
  n_likes = []
  n_shares = []
  n_comments = []
  n_all = []
  with open(fname) as f:
    for line in f:
      post = json.loads(line)
      created_time = dateutil.parser.parse(post['created_time'])
      n_likes.append(post['likes']['summary']['total_count'])
      n_comments.append(post['comments']['summary']['total_count'])
      try:
        n_shares.append(post['shares']['count'])
      except KeyError:
        n_shares.append(0)
      n_all.append(n_likes[-1] + n_shares[-1] + n_comments[-1])
      all_posts.append(created_time.strftime('%H:%M:%S'))
```

```
idx = pd.DatetimeIndex(all_posts)
data = {
  'likes': n_likes,
  'comments': n_comments,
  'shares': n_shares,
  'all': n_all
}
my_series = pd.DataFrame(data=data, index=idx)

# 重抽样成以 1 小时为区间
per_hour = my_series.resample('1h', how='sum').fillna(0)

# 画图
fig, ax = plt.subplots()
ax.grid(True)
ax.set_title("Interaction Frequencies")
width = 0.8
ind = np.arange(len(per_hour['all']))
plt.bar(ind, per_hour['all'])
tick_pos = ind + width / 2
labels = []
for i in range(24):
  d = datetime.now().replace(hour=i, minute=0)
  labels.append(d.strftime('%H:%M'))
plt.xticks(tick_pos, labels, rotation=90)
plt.savefig('interactions_per_hour.png')
```

可以用以下命令运行上述脚本。

```
$ python facebook_top_posts_plot.py --page PacktPub
```

matplotlib 的图像输出结果保存在 interactions_per_hour.png 文件中，如图 4-7 所示。

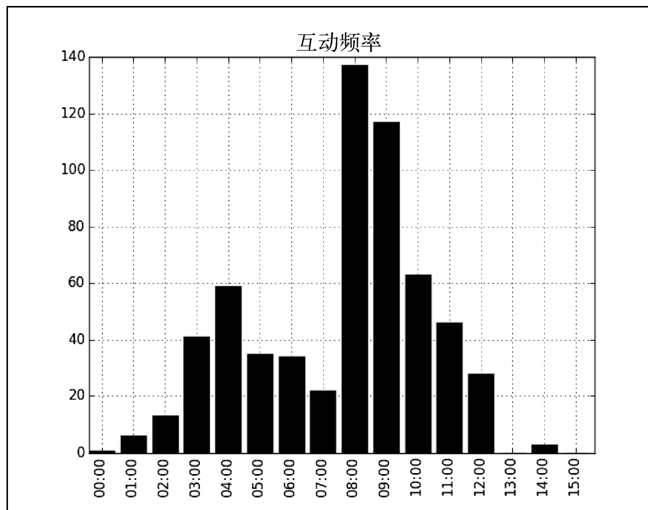


图 4-7　互动频率（每小时聚合）

代码迭代.jsonl 文件，为每条帖子构建列表以存储各种统计值：喜欢的数量、分享的数量、评论的数量以及它们的综合。这个列表的每一项是数据框中的一列，并用创建时间（仅用时间，而不用日期）索引。用前面介绍的重抽样技术来聚合 1 小时区间段内发布的所有帖子，然后对其频率求和。在这里的示例中，我们只考虑了互动的总次数，也可以画出单独的统计指标。

这幅图表明，最多的互动次数分布在 08:00 和 09:00。也就是说，在这种聚合下，发布于早上 8~10 点间的帖子会获得最多的互动次数。

观察数据集后，我们发现早上 8~10 点也是帖子发布最多的时间段。如果画出帖子的频率，可以观察到和图 4-7 类似的分布（这个留给你作为练习）。

这里的关键是聚合模式，即重抽样方法中的 `how` 属性。我们用的是 `sum`，早上 8~10 点具有最多的互动次数似乎是因为有更多的帖子可以互动。因此，加和并不能告诉我们这些帖子是否成功。解决方案很简单：这里不采用 `how='sum'`，可以将代码修改为 `how='mean'`，并重新运行代码。输出结果如图 4-8 所示。

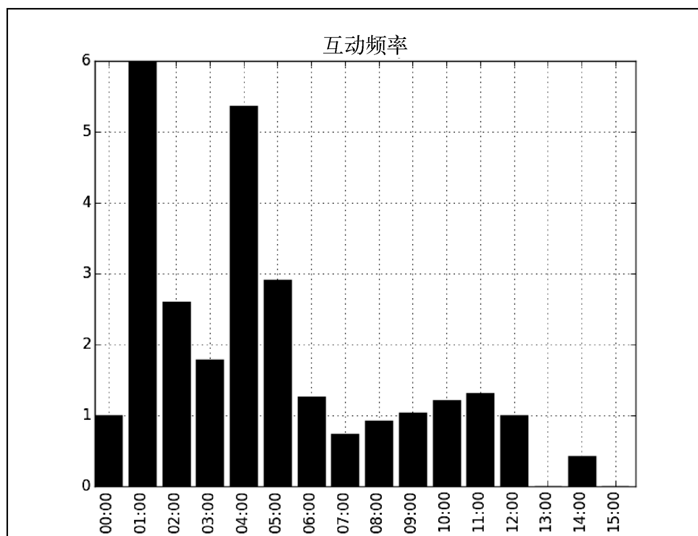


图 4-8　互动频率（每小时平均聚合）

图 4-8 展示了互动的平均次数的分布，以每小时的间隔聚合。在这幅图中，早上 8~10 点看起来并不像前面那样成功。最多的互动频率集中在凌晨 1 点和早上 5 点，分别有两个峰值。

这个简单的练习表明，用不同的方法聚合数据可以得到同一个问题的不同答案。进一步来说，需要特别注意的是，我们用的是同一批数据。首先，我们没有关于有多少人喜欢 PacktPub 页面的历史信息，即有多少人在其新闻订阅中看到 PacktPub 的帖子。如果很多人只是最近才关注这个页面，那么他们与较早帖子互动的可能性就不大，因此统计是偏向新帖子的。其次，我们并没

有关于人口统计的信息，也没有用户的地理信息：凌晨 1~5 点看起来有些奇怪，但正如前面提过的，生成时间被归一化为 UTC 时区。换句话说，凌晨 1~5 点对应的是印度的上午或者美国西海岸的傍晚或夜晚。

如果有关于用户的相关人口统计信息，就可以更好地理解发布帖子的最佳时间。例如，如果大多数用户位于美国西海岸，并且喜欢在傍晚（根据他们的时区）互动，那么将帖子的发布时间安排在 UTC 凌晨 1 点比较合适。

4.3.3　用词云可视化帖子

分析完互动后，我们将注意力转移到帖子的内容。

词云（或标签云）是文本数据的可视化表示。每个单词的重要程度由其在图中的大小表示。

本节将用 Python 的 wordcloud 包生成词云。首先需要用以下命令在虚拟环境中安装这个库及其依赖。

```
$ pip install wordcloud
$ pip install Pillow
```

Pillow 是 Python Imaging Library（PIL）项目的分支，而 PIL 已经不再使用了。除了继承 PIL 的功能，Pillow 还支持 Python 3，简单安装之后就可以使用了。

以下脚本首先读取存储 PacktPub 帖子内容的.jsonl 文件，然后创建一个词云的.png 图片。

```
# Chap04/facebook_posts_wordcloud.py
import os
import json
from argparse import ArgumentParser
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from wordcloud import WordCloud

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  fname = "posts_{}.jsonl".format(args.page)

  all_posts = []
  with open(fname) as f:
    for line in f:
      post = json.loads(line)
```

```
    all_posts.append(post.get('message', ''))
text = ' '.join(all_posts)
stop_list = ['save', 'free', 'today',
             'get', 'title', 'titles', 'bit', 'ly']
stop_list.extend(stopwords.words('english'))
wordcloud = WordCloud(stopwords=stop_list).generate(text)
plt.imshow(wordcloud)
plt.axis("off")
plt.savefig('wordcloud_{}.png'.format(args.page))
```

和前面一样，该脚本用 `ArgumentParser` 类的一个实例设置命令行参数（即页面名称或页面 ID）。

首先，脚本创建了一个列表 `all_posts` 来存储所有帖子的信息。之所以用 `post.get('message', '')` 而不是直接查询 Python 字典的值，是因为有的帖子中可能不存在 `message` 键（例如，一些图像没有评论内容），虽然这种情况非常少见，但仍然需要注意。

接着脚本将帖子列表合并成字符串 `text`，以作为生成词云的主要输入数据。`WordCloud` 对象接收一些可选参数来定义词云的样式。尤其值得注意的是，这个示例用 `stopwords` 参数定义词云会过滤的一组停用词。这里我们采用 NLTK 库中的常用英语停用词表以及一些自定义的关键词，后者常用于 PacktPub 账户信息且没有实际意义（如 bit.ly 和对特定标题的引用）。

输出图像如图 4-9 所示。

图 4-9　词云的示例

以上图像展示了 Packt 出版社的一些产品（纸质书、电子书和视频）及其出版物中讨论的一些主要技术（如 Python、JavaScript、R、Arduino、游戏开发等），主要关键词是**数据**。

4.4　小结

本章介绍了一些用于 Facebook 的数据挖掘应用，而 Facebook 是社会媒体领域的一大巨头。

介绍完 Facebook Graph API 及其演进，以及数据挖掘的概念后，我们创建了一个用于与 Facebook 平台交互的 Facebook 应用。

本章介绍的数据挖掘应用与鉴权用户的资料和 Facebook 页面相关。我们探讨了如何计算数值，这些数值与鉴权用户的发布习惯和用户关注给定页面的交互习惯有关。通过简单的聚合和可视化技术，可以用较少的代码实现这种分析。最后，我们介绍了如何用词云对重要的关键词进行可视化表示，以突出一组已发布帖子的重要主题。

下一章将重点介绍 Google+，它是最近出现的一个社交网络，由 Google 开发。

第 5 章 Google+话题分析

本章主要介绍 Google+（有时也称作 Google Plus 或 G+），它是社会媒体圈的新兴巨头。Google+于 2011 年面世，被描述为“架构在所有 Google 服务之上的社交层”。它的用户增长迅速，发布两个星期就获得了 1000 万用户。经过多次重新设计后，Google 于 2015 年年底发布了关注社区和集合的新版 Google+，将服务推向基于兴趣的网络。

本章将介绍以下主题：

- 如何在 Python 的帮助下与 Google+ API 互动
- 如何在 Google+中搜索人或者页面
- 如何用 Web 框架 Flask 在 Web GUI 中可视化搜索结果
- 如何从用户的发帖内容中抽取出有意义的关键词

5.1 Google+ API 入门

Google+ API 是 Google+的一个编程接口。该 API 可用于将应用或网站与 Google+整合起来，与前面介绍的 Twitter 和 Facebook 的情况类似。本节将介绍注册应用的过程，以及 Google+ API 入门。

如果还没有注册过 Google 账户，那么需要先注册一个。虽然 Google 提供了多种服务（如 Gmail、Blogger 等），但其账户管理是集中式的。也就是说，如果你是其中一种服务的用户，那么就可以快速地在 Google+中建立账户。

注册并登录后，你看到的是 Google 开发者控制台。可以从该控制台创建我们的第一个项目。图 5-1 展示了项目创建对话框，我们只需要指定项目的名称。

图 5-1　在 Google 开发者控制台中创建项目

创建好项目后，需要启用 Google+ API。在项目的面板中，Use Google APIs 组件允许我们管理 API 接入，创建新的用户凭证，等等。正如一个 Google 账户可用于获取多种 Google 服务，一个项目可以使用多个 API，只要这些 API 为启用状态。当在 Social APIs 组下定位到 Google+ API 时，我们可以单击鼠标来启动它。图 5-2 展示了概览。

图 5-2　为我们的项目启用 Google+ API

启动 API 后，系统会提醒我们，需要凭证信息才能使用该 API，如图 5-3 所示。

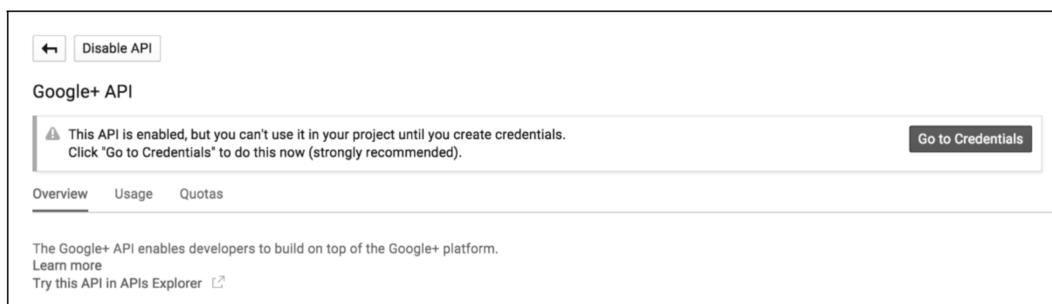


图 5-3　启动 API 后需要设置凭证

很容易在左侧边栏菜单中找到 Credentials 标签。凭证有三种：API key、OAuth client ID 和 Service account key（用 Google Cloud API 进行服务端到服务端的交互时，才需要使用 Service account key）。

简单的 API 访问需要 API key，也就是说，这种 API 调用不会获取任何私有用户数据。这个密钥启用应用级鉴权，主要用于统计项目的使用情况（后文将介绍有关接口访问频率限制的更多细节）。

OAuth client ID 用于需要授权的 API 访问，即需要获取私有用户数据的 API 调用。但在调用前，访问私有数据的用户必须向你的应用授权。不同的 Google API 声明了不同的使用范围，也就是必须获得用户许可才能执行的操作。

5

如果不太确定你的应用需要何种凭证，面板中还提供了 Help me choose 选项（如图 5-4 所示），通过一些简单的问题帮助你选择，这些问题有助于你明白自己的应用需要什么级别的权限。

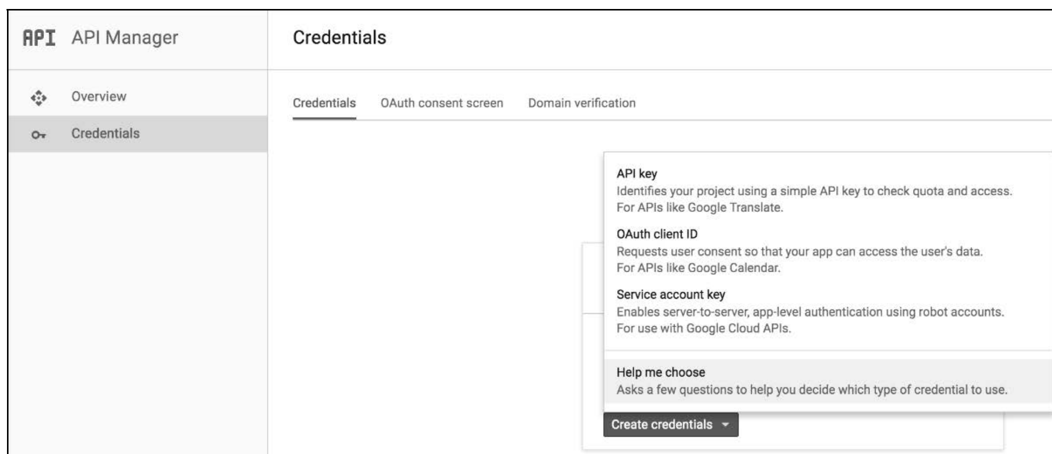


图 5-4　选择 API 访问凭证类型的下拉菜单

创建好项目并获得需要的访问密钥后，我们来看看如何以编程的方式访问 Google+ API。

Google 为 Google API 提供了一个官方的 Python 客户端，可以用 `pip` 在我们的虚拟环境中安装。

```
$ pip install google-api-python-client
```

可以在 `googleapiclient` 包（别名 `apiclient`）中调用该客户端。下面将介绍用于测试 API 使用情况的第一个示例。

在 Google+中搜索

`gplus_search_example.py` 脚本中显示的第一个示例查询用于搜索人和页面的 Google+ API。

这个示例假设你已经在项目面板的凭证页面中创建了可以进行简单访问的 API 密钥，现在还不需要获取私有用户数据。与我们在 Twitter 和 Facebook 中的做法类似，将凭证存储为环境变量。例如，如果使用 Bash 命令行，可以输入以下命令。

```
$ export GOOGLE_API_KEY="your-api-key-here"
```

也可以在一个 shell 脚本中包含 `export` 命令。

```
# Chap05/gplus_search_example.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--query', nargs='*')
  return parser

if __name__ == '__main__':
  api_key = os.environ.get('GOOGLE_API_KEY')
  parser = get_parser()
  args = parser.parse_args()

  service = build('plus',
                  'v1',
                  developerKey=api_key)

  people_feed = service.people()
  search_query = people_feed.search(query=args.query)
  search_results = search_query.execute()

  print(json.dumps(search_results, indent=4))
```

该脚本用 `ArgumentParser` 的一个实例从命令行读取查询字符串。例如，如果想要查询 `packt`，可以运行以下脚本。

```
$ python gplus_search_example.py --query packt
```

解析器的--query 参数定义了 nargs='*'属性，这样可以查询多项。

使用 API 的起点是先用 build()函数创建一个 service 对象。我们从 Google API 客户端导入服务构建器，它将以我们想要交互的服务的名称（plus）和 API 的版本（v1）为强制参数，再加上前面定义的 API 密钥 developerKey。



支持的 API 及其最新版本参见文档（https://developers.google.com/api-client-library/python/apis/）。

通常情况下，可以用以下方式使用服务构建器。

```
from apiclient.discovery import build
service = build('api_name', 'api_version', [extra params])
```

一旦创建完成，service 对象可以提供各种资源（分组为集合）的访问。这个示例查询的是 people 集合。

构建并执行搜索查询后，该脚本将 JSON 输出到屏幕，这样我们就可以理解这种格式。

```
{
  "nextPageToken": "token-string",
  "selfLink": "link-to-this-request",
  "title": "Google+ People Search Results",
  "etag": "etag-string",
  "kind": "plus#peopleFeed",
  "items": [
    {
      "kind": "plus#person",
      "objectType": "page",
      "image": {
        "url": "url-to-image"
      },
      "id": "112328881995125817822",
      "etag": "etag-string",
      "displayName": "Packt Publishing",
      "url": "https://plus.google.com/+packtpublishing"
    },
    {
      /* 更多项…… */
    }
  ]
}
```

出于简洁的目的，我们简化了以上输出，但大体结构还是清楚的。第一层的属性（如 title、selfLink 等）定义了结果集合的一些特性。结果列表包含在 items 列表中。每一项由 id 属性唯一标识，并由各自的 url 表示。这个示例展示的结果可包含 objectType 属性中定义的页面和人。displayName 属性由用户（或页面管理者）选择，且通常显示在相应的 Google+页面中。

5.2　在 Web GUI 中嵌入搜索结果

本节将扩展第一个示例，以便对搜索结果进行可视化。为了展示各项，我们将用一个自动动态生成的网页，以使用熟悉的接口将资料图片显示在每个人或页面的显示名称旁边。这样的可视化可以帮助我们区别具有相同显示名称的不同结果。

为了实现这个目标，我们将介绍用于 Web 开发的微框架 Flask，它可以帮助我们快速生成一个 Web 界面。

广义上说，Python 和 Web 开发是共同发展的，与 Web 相关的几个 Python 库已问世多年，达到了一定的成熟度。与其他框架相比，Flask 非常年轻，但也达到了一定的成熟度，而且被广大社区采用。由于 Web 开发并非本书重点，我们采用 Flask 主要是看重其微的特性，即不对应用结构做出假设，并且使用少量代码就可以轻松实现 Web 开发。

Flask 的一些特征如下：

- 开发服务器和调试器
- 整合了对单元测试的支持
- 支持使用 Jinja2 库的模板
- 基于 Unicode
- 具有适用于各种场景的大量扩展

可以通过以下命令用 `pip` 安装 Flask。

```
$ pip install flask
```

以上命令将安装微框架和相关的依赖。

如果感兴趣，可以在 Packt 出版社找到很多介绍 Flask 的图书来进一步学习。例如，Matt Copperwaite 和 Charles Leifer 撰写的 *Learning Flask Framework* 以及 Jack Stouffer 撰写的《深入理解 Flask》都介绍了 Flask 的很多高级用法。本章不会过多介绍 Flask，以下是使用 Flask 的一个示例。

```
# Chap05/gplus_search_web_gui.py
import os
import json
from flask import Flask
from flask import request
from flask import render_template
from apiclient.discovery import build

app = Flask(__name__)
api_key = os.environ.get('GOOGLE_API_KEY')

@app.route('/')
```

```
def index():
  return render_template('search_form.html')

@app.route('/search', methods=['POST'])
def search():
  query = request.form.get('query')
  if not query:
    # 如果未给出查询语句，则显示错误消息
    message = 'Please enter a search query!'
    return render_template('search_form.html', message=message)
  else:
    # 搜索
    service = build('plus',
                    'v1',
                    developerKey=api_key)

    people_feed = service.people()
    search_query = people_feed.search(query=query)
    search_results = search_query.execute()
    return render_template('search_results.html',
                           query=query,
                           results=search_results['items'])

if __name__ == '__main__':
  app.run(debug=True)
```


gplus_search_web_gui.py 脚本用 Flask 实现了一个基本的 Web 应用，展示了一个查询 Google+ API 的简单表单，并显示了结果。

该应用是 Flask 类的一个简单实例，其主函数是用于启动 Web 服务器的 run()函数（我们将在调试模式中运行，以简化调试过程）。

Web 应用的行为由其路由定义。也就是说，当执行一个特定 URL 请求时，Flask 会将该请求分发到相应的代码段，然后生成响应。在 Flask 中，路由是简单的装饰函数。

5.2.1 Python 的装饰器

虽然在 Python 中使用装饰器非常简单，但如果你没有接触过装饰器这个概念，理解起来还是会有些困难。

装饰器就是一个函数，可作为另一个函数的封装器来丰富它的行为。这种行为的改变是动态的，因为并不需要对目标函数代码进行任何改变，也不需要使用子类。这样，就可以在不对现有代码引入任何复杂性的前提下改进特定功能。

总的来说，装饰器是一种强大而优雅的工具，在很多场景中都非常实用。

在前面的示例中，index()和 search()都是装饰函数，且装饰器是 app.route()。该装饰器紧接在目标函数前调用，并以@符号作为前缀。

5.2.2 Flask 路由和模板

app.route()装饰器以我们要访问资源的相对 URL 为第一个参数。第二个参数是特定 URL 支持的 HTTP 方法列表。如果没有给出第二个参数，那么默认是 GET 方法。

index()函数用于展示进入页面，该页面包含搜索表单，用相对 URL /表示。该函数所做的就是通过 render_template()函数返回存储在 search_form.html 模板中的网页。

由于 Flask 使用的是 Jinja2 模板库，render_template()函数读取存储在 HTML 文件中的代码，应用模板指令，并返回最终的 HTML 页面作为输出。

这里我们不过多介绍细节，如果感兴趣，可以在官方文档中查看更多精彩内容。使用模板库的目的是介绍一个嵌入网页的特殊语法，可以解析该网页来动态生成 HTML。

templates/search_form.html 源文件存放在 Python 运行 Flask（示例中是 gplus_search_web_gui.py 脚本）的相同路径中，该 HTML 文件的内容如下所示。

```
<html>
  <body>

    {% if message %}
    <p>{{ message }}</p>
    {% endif %}

    <form action="/search" method="post">
      Search for:
      <input type="text" name="query" />
      <input type="submit" name="submit" value="Search!" />
    </form>

  </body>
</html>
```

上述源文件包含一个带有表单的基本网页（为了简洁进行了简化）。该页面的唯一模板指令是 if 模块，该模块检查 message 变量；如果存在，就将其显示出来。

用 POST 方法将表单动作设置为相对 URL /search。这是 gplus_search_web_gui.py 文件中的第二个装饰器函数 search()的配置。

search()函数是与 Google+ API 进行交互的地方。首先，该函数期望一个 query 参数通过表单传递进来。可以通过 request.form 字典获取该参数。

Flask 中的全局 request 对象用于获取传入的请求数据。request.form 字典用于获取数据，而数据是用 POST 方法通过一个表单传递的。

如果没有给出查询，那么 search()函数将再次显示搜索表单，并给出一个错误消息。

render_template()函数读入一些关键字参数，并将它们传递给模板，我们的示例中只有一个参数 message。

另一方面，如果给出了查询，search()函数将与 Google+ API 交互，并将交互结果传递给 search_results.html 模板。templates/search_results.html 的源文件如下所示。

```
<html>
  <link rel="stylesheet"
    href="{{ url_for('static', filename='style.css') }}" />
  <body>

    Searching for <strong>{{ query }}</strong>:

    {% for item in results %}
    <div class="{{ loop.cycle('row_odd', 'row_even') }}">
      <a href="{{ item.url }}">{{ item.displayName }}</a>
      ({{ item.objectType}})<br />
      <a href="{{ item.url }}">
        <img src="{{ item.image.url }}" />
      </a>
    </div>
    {% endfor %}

    <p><a href="/">New search</a></p>

  </body>
</html>
```

5

整段代码的核心是一个迭代结果列表的 for 循环。列表中的每一项都显示了一个<div>元素及其内容信息。我们注意到，打印特定变量值的语法由一对花括号构成，例如，{{ item.displayName }}会打印每项的显示名称。另一方面，控制流包含在一对{%和%}符号中。

这个示例还将 CSS 整合到 HTML 文件中，这些 CSS 用于个性化设置结果页面的外观。loop.cycle()函数用于在字符串列表或变量列表中进行循环，这里它将不同的 CSS 类分配给结果中的<div>代码块。这样一来，就可以用不同的颜色高亮显示不同的行。此外，url_for()函数用于为特定的资源提供 URL。与 templates 文件夹类似，static 文件夹也必须与运行 Flask 应用的 gplus_search_web_gui.py 文件放在同一个文件夹中。它用于提供静态文件，如图像或 CSS 定义。static/style.css 文件包含用于搜索结果页面的 CSS 的定义，如下所示。

```
.row_odd {
  background-color: #eaeaea;
}
.row_even {
  background-color: #fff;
}
```

虽然 Web 开发与数据挖掘并非密切相关，但它提供了一种快速构建简单 UI 原型的方式。Web 开发的主题非常广泛，本章不做详细介绍。如果感兴趣，你可以进一步深入了解 Web 开发和 Flask。

可以用以下命令运行上面的示例。

```
$ python gplus_search_web_gui.py
```

上述命令运行后会开启 Flask 应用，而 Flask 应用将等待 HTTP 请求，以提供 Python 代码中定义的响应。运行脚本后，该应用将在前端运行，终端将显示以下输出。

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

`127.0.0.1` 地址是本地主机，而 `5000` 端口是 Flask 的默认端口。



Flask 服务器的网络可见性

如果在虚拟机或者远程主机上运行 Flask 应用，那么可能需要该机器的网络地址，而不是本地主机的地址。

此外，你还需要确保该服务器是外部可见的，这样才能通过网络路由到该服务器。当然，这会带来安全问题，因此建议你先考虑是否信任你的网络用户。

为了使 Flask 应用在外部可见，你可以停用调试模式或者捆绑服务器和特定的地址，如 `app.run(host='0.0.0.0')`。这里的 `0.0.0.0` 意味着**本机的所有地址**。

现在我们可以打开一个浏览器窗口，输入 http://127.0.0.1:5000/，如图 5-5 所示。

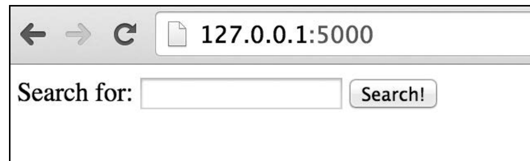


图 5-5　Flask 应用的进入页面

如果不输入任何内容，直接点击 Search!按钮，那么该应用将重新展示表单，并给出一个错误消息，如图 5-6 所示。

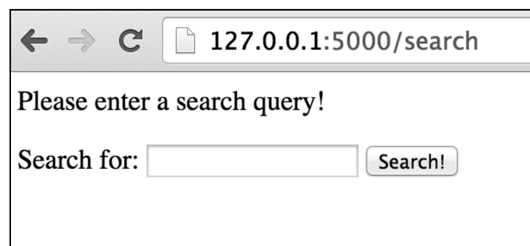


图 5-6　没有给出查询则显示错误消息

另一方面，如果给出正确的查询，结果页面将如图 5-7 所示（图中的查询是 `packt`）。

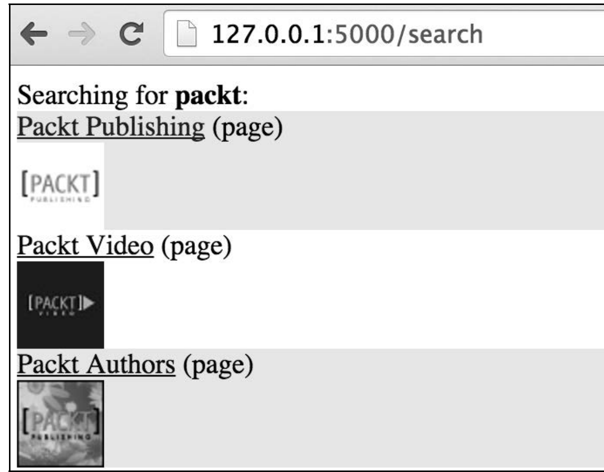


图 5-7 在 Google+ API 中搜索的结果

正如我们看到的，每项都在自己的显示块中显示，并且每个显示块带有不同的颜色及相应项的信息。项的名称在锚定标签（<a>）中表示。这样一来，这些项就是可点击的，并可以链接到相关的 Google+页面。

5.3 Google+页面的笔记和活动

搜索 Google+页面并在 Web GUI 中可视化结果后，我们将下载给定页面的活动列表。Google+中的活动等同于 Facebook 中的帖子。默认情况下，一个活动可以当作一个笔记，即在 Google+上分享的一段文字。

以下脚本 gplus_get_page_activities.py 用于从一个 Google+页面搜集活动列表。

```
# Chap05/gplus_get_page_activities.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--page')
  parser.add_argument('--max-results', type=int, default=100)
  return parser

if __name__ == '__main__':
  api_key = os.environ.get('GOOGLE_API_KEY')
  parser = get_parser()
  args = parser.parse_args()
```

```
service = build('plus',
                'v1',
                developerKey=api_key)

activity_feed = service.activities()
activity_query = activity_feed.list(
                                    collection='public',
                                    userId=args.page,
                                    maxResults='100'
                                    )
fname = 'activities_{}.jsonl'.format(args.page)
with open(fname, 'w') as f:
  retrieved_results = 0
  while activity_query and retrieved_results < args.max_results:
    activity_results = activity_query.execute()
    retrieved_results += len(activity_results['items'])
    for item in activity_results['items']:
      f.write(json.dumps(item)+"\n")
    activity_query = service.activities().list_next(activity_query,
                                                     activity_results)
```

该脚本用 `ArgumentParser` 从命令行获取一些输入参数。`--page` 选项是必需的，因为它包含了我们寻找的页面或用户的 ID，既可以是页面的数值 ID，也可以是 Google+句柄（例如，`+packtpublishing` 就是 Packt 出版社的 G+页面）。另一个参数是我们想要检索的结果（即活动）的最大数量。该参数是可选的，并且默认为 `100`。

可以用以下方式运行该脚本。

```
$ python gplus_get_page_activities.py --page +packtpublishing
  --max-results 1000
```

几秒后，我们将在 activities_+packtpublishing.jsonl 文件中看到 JSON 格式的活动列表。与使用 Twitter 和 Facebook API 生成的文件一样，该文件也是 JSON Lines 格式的，即文件的每一行都是一个有效的 JSON 文档。

单个活动的详细描述可以查看文档（https://developers.google.com/+/web/api/rest/latest/activities#resource）。下面列出了最重要的字段。

- `kind`：项的类型（即 `plus#activity`）
- `etag`：实体标签字符串
- `title`：活动的短标题
- `verb`：一次发布或分享
- `actor`：一个对象，表示分享或发布活动的用户
- `published`：ISO 8601 格式的发布日期
- `updated`：ISO 8601 格式的最新更新日期

- ❑ id：活动的唯一标识符
- ❑ url：活动的 URL
- ❑ object：包含活动所有信息的一个复杂对象，其中包括活动的内容、原始发布者（如果活动是分享，那么发布者与分享者不是同一个人）、回复的信息、分享和+1、附件列表和相关细节（如图像信息），还可能包含地理位置信息

对象的描述引入了+1 的概念，它与 Facebook 的**喜欢**按钮功能类似。当用户喜欢 Google+上的某条内容时，可以点击+1 按钮。+1 按钮允许用户分享 Google+上的内容，并且会推荐 Google 搜索上的相应内容。可以在 JSON 文档的 plusoners 键中找到特定对象的+1 内容的详情。plusoners.totalItems 包含+1 动作的数量，而 plusoners.selfLink 包含到+1 列表的一个 API 链接。类似地，replies 和 resharers 包含直接评论和分享。

查看下载活动列表的代码的逻辑，我们可以看到，起点仍然是通过 apiclient.discovery.build()函数构建服务对象。接下来将活动集合存储在 activity_feed 对象中。为了查询 activity_feed 对象，我们将使用它的 list()函数而不是 search()函数，因为要检索完整的活动列表（至少是可获得结果的最大数量）。该函数需要两个强制参数：collection 和 userId，其中 collection 只接受 public 作为值，userId 参数是我们正在检索的活动所属的用户或页面的标识符。该参数可以是唯一数值标识符或 Google+句柄字符串。可以将 maxResults 作为一个可选参数，在示例中设置为 100。这是通过一个 API 调用能够检索的项的最大数量（默认为 20，最大为 100）。

上述代码还展示了如何用这个 API 翻页。查询是在 while 循环中执行的，它会更新检索结果的计数器（循环+1 至--max-results 指定的值），并检查下一个页面的结果是否存在。list_next()函数以当前查询和结果的当前列表为参数，通过更新查询对象进行翻页。如果下一个结果页面不存在（即用户或页面只发布了当前我们已经检索到的这些帖子），则该函数会返回 None。

5.4 笔记的文本分析和 TF-IDF 计算

探讨完如何下载特定页面或用户的笔记列表和活动列表后，我们将关注点转移到内容的文本分析。

我们想要从给定用户发布的每条帖子中抽取有意义的关键词，以便对帖子本身做摘要或总结。

虽然直觉上这是一个简单的任务，但还需要考虑一些细微之处。实际上，我们可以发现每条帖子的内容并不总是一段干净的文本，其中可能包含 HTML 标签。在计算之前，我们需要抽取出干净的文本。虽然 Google+ API 返回的 JSON 对象具有清晰的结构，但内容本身并不是结构清晰的文档。好在有一个非常好的 Python 包可以帮忙。Beautiful Soup 可以解析 HTML 和 XML 文

档，也可以解析一些畸形的标记。它是兼容 Python 3 的，可以通过 CheeseShop 安装。在我们的虚拟环境中，使用以下命令安装 Beautiful Soup。

```
$ pip install beautifulsoup4
```

以上命令会安装 4.*版本。Beautiful Soup 从版本 3.*到版本 4.*经历了一些重大调整，因此本书中的部分示例可能不兼容该库的老版本。

下一个重要问题是，如何定义一个关键词的重要程度?

可以从不同方向解决该问题，而重要的定义也会根据应用发生变化。这里，我们将使用基于统计的方法，关键词的重要程度由其在文档和文档集合中的出现频率决定。

我们使用的方法叫作 TF-IDF，它是基于频率的两个分数值（TF 和 IDF）的组合。在第 3 章中，我们通过 scikit-learn 库提供的现成实现介绍了 TF-IDF。总的来说，使用现成的实现是一个好主意，尤其是在其来自 scikit-learn 这样高质量的库时。本节，我们将提出一个自定义的实现，这样就可以展示 TF-IDF 的详情，以便你更好地理解该框架。

TF-IDF 背后的动机非常简单：如果一个单词是一篇文档中的高频词，但在所有文档集合中是一个低频词，那么它就是能够表示该文档的一个候选。这两个性质通过两个分数值来反映：TF 表示一个词在一篇文档中出现的频率，IDF 计算整个文档集合。

1972 年，Karen Sparck-Jones 在信息检索研究中提出了 IDF，他是该领域的先锋人物之一。作为一种启发式方法，IDF 显示了指定项和 Zipf 定律间的关系。



Zipf 定律和 Zipf 分布

Zipf 定律是一个经验定律，指的是不同科学领域中的多种类型的数据都可以近似为 Zipf（即长尾）分布。在第 2 章中，我们观察到 Packt 出版社在推文中使用的单词遵循该分布。有关 Zipf 定律的更多细节，参见第 2 章。

从数学的角度来看，这些年提出了 TF 和 IDF 的很多变体。

两种最常用的 TF 方法是，只考虑一个单词在一篇文档中的原始频率，以及用一篇文档中的单词总数对单个单词进行归一化（即观察该单词在一篇文档中的出现概率）。

在 IDF 方面，传统的定义是 `log(N/n)`，其中 `N` 是文档的总数，`n` 是包含给定单词的文档数目。对于出现在每篇文档中的单词，其 IDF 的值将是 0（即 `log(1)`）。因此，IDF 的另一个归一化方法是 `1+log(N/n)`。

图 5-8 展示了 TF-IDF 及其与 Zipf 定律的关系，即出现频率太高或太低的单词都没有代表性。

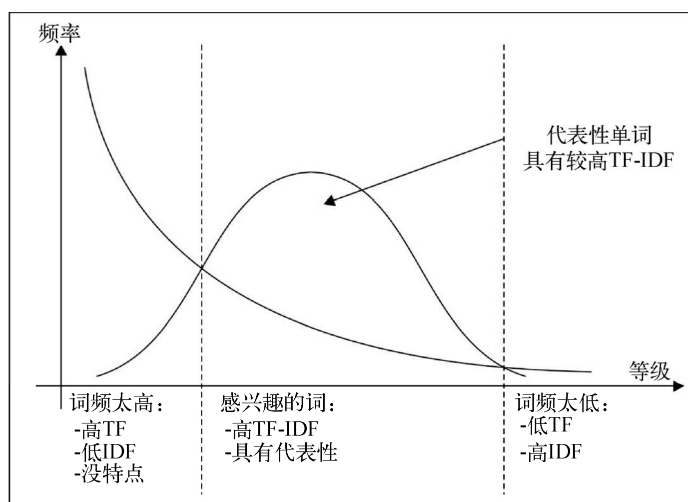


图 5-8　单词及其重要程度的分布

高频单词中会包含停用词，如冠词、连词和介词。通常来说，这些词本身不具有任何含义，因此可以忽略（即删除）。另一方面，那些罕见的单词主要是拼写错误的单词以及专业词汇。取决于应用，探索数据集可以更好地了解移除停用词是否是个好主意。

gplus_activities_keywords.py 脚本从给定的 JSON Lines 文件中读取活动列表，计算每个活动的文本内容中每个单词的 TF-IDF 值，并将最高频的关键词显示在每条帖子的旁边。



该代码用 NLTK 库来执行一些文本处理操作，如分词和移除停用词。我们在第 1 章中探究了如何下载需要用到的 NLTK 包，特别是用于分词的 punkt 包。

gplus_activities_keywords.py 脚本如下所示。

```
# Chap05/gplus_activities_keywords.py
import os
import json
from argparse import ArgumentParser
from collections import defaultdict
from collections import Counter
from operator import itemgetter
from math import log
from string import punctuation
from bs4 import BeautifulSoup
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

我们定义了一个停用词列表，该列表由常用英语停用词和标点符号组成。

```
punct = list(punctuation)
all_stopwords = stopwords.words('english') + punct

def get_parser():
```

```
    parser = ArgumentParser()
    parser.add_argument('--file',
                        '-f',
                        required=True,
                        help='The .jsonl file with all the activities')
    parser.add_argument('--keywords',
                        type=int,
                        default=3,
                        help='Number of keywords to extract for each post')
    return parser
```

预处理步骤由 `preprocess()` 函数处理，主要包括归一化、数据清洗（使用 `clean_html()` 辅助函数）和分词。

```
def clean_html(html):
    soup = BeautifulSoup(html, "html.parser")
    text = soup.get_text(" ", strip=True)
    text = text.replace('\xa0', ' ')
    text = text.replace('\ufeff', ' ')
    text = ' '.join(text.split())
    return text

def preprocess(text, stop=all_stopwords, normalize=True):
    if normalize:
        text = text.lower()
    text = clean_html(text)
    tokens = word_tokenize(text)
        return list(bigrams(tokens))
        # return [tok for tok in tokens if tok not in stop]
```

以下函数处理单词的统计。`make_idf()` 创建一个 IDF 分值集合，`get_keywords()` 函数用其计算每个文档的 TF-IDF 分数值，以提取有趣的关键词。

```
def make_idf_matrix(corpus):
    df = defaultdict(int)
    for doc in corpus:
        terms = set(doc)
        for term in terms:
            df[term] += 1
    idf = {}
    for term, term_df in df.items():
        idf[term] = 1 + log(len(corpus) / term_df)
    return idf

def get_keywords(doc, idf, normalize=False):
    tf = Counter(doc)
    if normalize:
        tf = {term: tf_value/len(doc)
              for term, tf_value in tf.items()}
    tfidf = {term: tf_value*idf[term]
               for term, tf_value in tf.items()}
    return sorted(tfidf.items(),
                  key=itemgetter(1),
```

```
                reverse=True)

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  with open(args.file) as f:

    posts = []
    for line in f:
      activity = json.loads(line)
      posts.append(preprocess(activity['object']['content']))

  idf = make_idf_matrix(posts)

  for i, post in enumerate(posts):
    keywords = get_keywords(post, idf)
    print("----------")
    print("Content: {}".format(post))
    print("Keywords: {}".format(keywords[:args.keywords]))
```

和前面的示例一样，我们用 `ArgumentParser` 读取命令行参数。`--file` 用于传递文件名，可选参数`--keywords` 指定每篇文档中我们希望关注的关键词个数（默认为 `3`）。

可以用以下命令运行上述脚本。

```
$ python gplus_activities_keywords.py --file
  activities_+LarryPage.jsonl --keywords 5
```


该命令会为用户+LarryPage 读取活动的.json1 文件（假设已经提前下载好），并显示每个活动的前 5 个关键词。

例如，Larry Page 的其中一条帖子如下所示。

```
----------
Content: ['fun', 'day', 'kiteboarding', 'alaska', 'pretty', 'cold',
'gusty', 'ago']
Keywords: [('alaska', 5.983606621708336), ('gusty', 5.983606621708336),
('kiteboarding', 5.983606621708336), ('cold', 4.884994333040227),
('pretty', 3.9041650800285006)]
```

正如我们所见，帖子以各项列表的形式呈现，关键词以元组列表的形式呈现，且每个元组包含词项及其 TF-IDF 分值。

从全文本到词项列表的转变由 `preprocess()`函数处理，该函数将 `clean_html()`函数用作辅助函数。

预处理过程会执行以下步骤：

- ❑ 文本归一化（即全部转换为小写）
- ❑ HTML 清洗/去除

- ❑ 分词
- ❑ 停用词移除

preprocess()函数只需要一个强制参数，我们可以自定义要移除的停用词列表（若不想执行这个步骤，则可以传入一个空列表），也可以通过设置关键字参数 stop 和 normalize 来关闭小写转换。

文本归一化用于合并只是大小写形式不同的单词。例如，单词 The 和 THE 通过小写处理会映射到 the。小写转换在很多应用中都是有意义的，但在有些情景并不适用，如 us（代词）和 U.S.（United States 的缩写）并不相同。注意，小写转换不可逆，因此，如果需要在后面的应用中参考原始文本，那么还需要将其存储下来。

用 Beautiful Soup 清洗 HTML 只需要几行代码。BeautifulSoup 对象利用 html.parser 作为第二个参数实例化后，就可以处理畸形的 HTML。get_text()函数只获取文本，将 HTML 标签留下。作为这个函数的第一个参数，空格用于替换被去除的 HTML 标签。它适用于以下情景。

```
... some sentence<br />The beginning of the next one
```

在以上示例中，只去除断行标签会生成一个 sentenceThe 词项，这显然不是一个真实的单词。用一个空格替换这个标签就可以解决这个问题。

clean_html()函数还用几个字符串替换空白的 Unicode 字符。最后，如果文本中有很多连续的空白，会通过分割文本然后再拼接起来（如' '.join(text.split())）归一化为一个空白。

分词是将字符串分割为独立词项的过程，由 NLTK 的 word_tokenize()函数处理。

最后，用一个简单的列表推导式实现停用词移除，检查每个词项是否出现在 stop 列表中。NLTK 的英语停用词列表和 string.punctuation 的标点符号列表合并为默认的停用词列表。

TF-IDF 由两个函数实现：make_idf()和 get_keywords()。由于 IDF 是基于整个文档集合的统计，我们会首先计算它。corpus 参数是一个列表的列表，即一个分词文档的列表，这些文档已经由 preprocess()函数处理过了。

首先，我们用 df 字典计算文档频率。它是出现了某个单词的文档总数，也就是说不考虑单词重复出现的情况。对于语料库中的每篇文档，我们会将其转换为一个集合。这样一来，文档中的每一项都是唯一的。其次，我们将迭代新构建的 df 字典的各项，并用其计算 IDF 值，然后由该函数返回。就这个版本的 IDF 而言，我们将使用前面提到的+1 归一化。

至此，我们可以着手计算文档中每一项的 TF-IDF 值了。调用 get_keywords()函数来迭代所有帖子，该函数需要两个参数：文档（词项列表）和前面构建的 IDF 字典。第三个可选参数是一个标记，用于激活文档长度的归一化步骤，这会有效地将 TF 值转化为词项在文档中出现的概率，即 P(t|d)。

`get_keywords()`函数首先通过 `collections.Counter` 创建一个原始的 TF 值，它本质上是一个字典。如果文档长度需要归一化，则通过一个字典推导式重建 `tf` 字典，计算方法是将每个原始的 TF 值除以文档中项的总数（即文档的长度）。

将每一项的 TF 值和 IDF 值相乘来计算 TF-IDF 值。这里再次使用了字典推导式。

最后，该函数返回项以及相应 TF-IDF 值的列表，并按照值的大小排序。用 `key= itemgetter(1)` 调用 `sorted()`对 TF-IDF 值（元组的第二项）进行排序，并用 `reverse=True` 进行降序显示。

用 n-gram 方法捕获短语

TF-IDF 对关键词在一篇文档中的重要程度提供了简单的数值统计。使用该模型时，每个单词是单独计算的，单词的顺序无关紧要。实际上，单词的使用是有先后顺序的，而且单词序列在句子结构中通常是一个独立单元（也称作语言学中的组成成分）。

给定一个分词后的文档（即一个词项序列），我们称 n-gram 是该文档中 n 个邻近项的序列。n=1 时，我们考虑的仍是单个项（也称作 unigram）。当 n 大于 1 时，我们考虑的是任意长度的序列。典型的示例是 bigram（n=2）和 trigram（n=3），但任意长度的序列都是有可能的。

给定词项的输入列表，NLTK 提供了快速计算 n-gram 列表的方法。NLTK 用 `nltk.util.ngrams` 函数处理 n-gram 的生成。

```
>>> from nltk.util import ngrams
>>> s = 'the quick brown fox jumped over the lazy dog'.split()
>>> s
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
>>> list(ngrams(s, 4))
[('the', 'quick', 'brown', 'fox'), ('quick', 'brown', 'fox', 'jumped'),
('brown', 'fox', 'jumped', 'over'), ('fox', 'jumped', 'over', 'the'),
('jumped', 'over', 'the', 'lazy'), ('over', 'the', 'lazy', 'dog')]
```

我们可以看到，该函数需要两个参数：一个词项序列和一个数字。`ngrams()`返回的值是一个生成器对象，因此，为了在命令行中显示出来，我们将其转换为一个列表，但也可以直接在一个列表推导式中使用它。

NLTK 还提供了两种快捷方法：`nltk.bigrams()`和 `nltk.trigrams()`。它们可以分别用 n=2 和 n=3 调用 `ngrams()`。

如果感兴趣，你可以思考以下问题。

- 能否修改本节中给出的代码来捕获有意义的 bigram 或 trigram，而不只是单个单词？
- 当捕获 n-gram 时，移除停用词有什么影响？

5.5 小结

本章介绍了 Google+ API，还讨论了如何在 Google 开发者控制台注册一个项目，以及如何启用项目需要用到的 API。

本章首先用一个示例展示了如何用 Google+ API 进行搜索。然后基于这个示例，介绍了如何将 API 嵌入 Web 应用。我们用 Flask 这个 Web 开发的微架构创建了一个 Web GUI 来显示搜索会话的结果。

接下来分析了一个用户或页面的文本笔记。下载用户活动并将其存储为 JSON Lines 格式后，我们介绍了 TF-IDF 这种从文本中抽取有趣关键词并进行数值统计的方法。

下一章会将注意力转移到问题回答领域，我们将探索 Stack Overflow API 的使用。

第 6 章

Stack Exchange 提问和回答

本章关于问答网络 Stack Exchange，以及问题回答这个更广泛的主题。

本章将介绍以下主题：

- 如何创建一个与 Stack Exchange API 互动的应用
- 如何在 Stack Exchange 搜索用户和问题
- 如何为线下数据分析导出数据
- 用于文本分类的监督机器学习
- 如何用机器学习预测问题标签
- 如何将机器学习模型嵌入实时应用

6.1 提问和回答

查找特定信息的答案是 Web 的一个主要用途。随着时间的推移，该技术在不断发展，互联网用户也在不断改变其在线行为。

人们在互联网上查找信息的方法与 15~20 年前大不相同。在过去，查找答案主要指使用搜索引擎。Amanda Spink 等人于 2001 年撰写的 *Searching the Web: The Public and Their Queries* 一书关于 20 世纪 90 年代后期的主流搜索引擎查询结果。这本书表明，那个年代的搜索一般非常短（平均 2.4 个词）。

最近几年，我们开始经历从短的基于关键词的搜索到长的对话式搜索（或问题式搜索）的转变。换句话说，搜索引擎已经从关键词匹配转移到**自然语言处理**。例如，图 6-1 显示了 Google 如何自动补全来自用户的查询/问题。该系统用流行的查询数值统计信息来预测用户的真实问题。

自然语言的 UI 为最终用户提供了更自然的体验。其中一个更先进的示例就是嵌入智能手机应用的智能私人助理。例如，Google Now、Apple 的 Siri 和 Microsoft 的 Cortana 都允许用户搜索 Web，得到有关餐馆的推荐或者帮助安排约会行程。



图 6-1　Google 的一个查询自动补全示例

提出问题只是工作的一部分，最重要的是找到问题的答案。图 6-2 显示了 Google 根据查询给出的结果页面：why is python called python。当 Google 将这个查询编译为一个常规查询，即一个问题而不是通常的信息查询时，就会激活所谓的**答案框**。该答案框包含来自一个网页的片段，Google 认为该片段就是给定问题的答案。页面的其余部分没有展示在图片中，包含的是传统的**搜索引擎结果页面**。

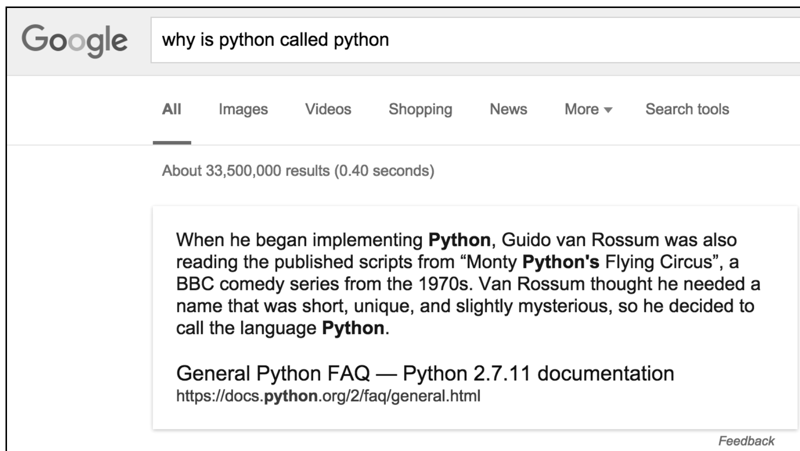


图 6-2　答案框作为对话式查询结果的一个示例

关于自动**问答**（QA）的研究并不局限于搜索引擎领域。IBM 的 Watson 是人工智能领域非常知名的一项成就，它是 IBM 公司开发的问答系统，并以 IBM 的首任 CEO 的名字 Thomas Watson 命名。该问答机器人完成时就获得了广泛关注，而且在游戏 Jeopardy!中打败了人类冠军。Jeopardy!是一个非常流行的美国问答游戏节目，参赛选手必须在给出问题的相关线索后回答出问题的答案。尽管该游戏与上述过程（你必须找到给定答案的正确问题）相反，但是解决这一问题所需的技术是可以双向操作的。

虽然搜索引擎和智能私人助理在不断改善用户在互联网上查找信息的方式，但用户迫切找到答案的需求促进了问答网站的崛起。最著名的问答网络是 Stack Exchange，目前包含很多专题网站：从技术到科学、从商业到娱乐，其中最知名的是 Stack Overflow，里面有各种有关编程的大

量问题和回答。Stack Exchange 的其他网站还包括 Movies & TV（服务于电影和电视节目爱好者）、Seasoned Advice（服务于专业和业余厨师）、English Language & Usage（服务于语言学家、语源学家和英语爱好者）和 Academia（服务于学者和接受过高等教育的人群）。Stack Exchange 的网站列表非常长，包含的主题也多种多样。

Stack Exchange 成功的原因之一可能是该社区的内容质量非常高。因为问题和回答都要接受用户投票，所以高质量的回答会上升至靠前的位置。用户通过使用网站挣得声望点的机制使得我们可以识别社区中最活跃的成员以及他们的专业。用户在他们的回答获得**赞**或**踩**投票时会获得或失去相应的声望点数。系统的游戏化还包括用户可以通过贡献获得的一系列**勋章**。

为了保持内容的高质量，重复的问题和低质量的回答通常会被截留，由审核人员复核，并最终被修改或关闭。

社交网络服务的一些主要特征并未在 Stack Exchange 中体现。例如，这里不可能直接与其他用户连接（如 Facebook 和 Twitter 中的好友或粉丝关系），也不可能与其他用户进行私人对话。不过 Stack Exchange 作为一个社会媒体平台的效果还是非常清晰的，即用户可以通过它交换知识并协作。

下一节将介绍 Stack Exchange API。

6.2 Stack Exchange API 入门

Stack Exchange API 提供了接入所有 Stack Exchange 网站的可编程接口。想要使用该 API 的第三方应用必须在 https://stackapps.com/注册以获得一个请求键，该键用于授权每天的更多请求。注册应用也必须执行鉴权的 API 调用，即以鉴权用户的身份与 Stack Exchange 交互。注册过程非常简单，只需要 Application Name 和 Description，如图 6-3 所示。

图 6-3 stackapps.com 的应用注册页面

注册应用的直接效果就是 API 允许的每日请求次数上限会增加。需要注意的是，访问频率限制不只限制每日访问量，而且禁止了网络流量攻击（例如，如果一个 IP 每秒发送超过 30 次请求，那么这些请求会被丢掉）。有关访问频率限制的详细信息可参见官方文档（https://api.stackexchange.com/docs/throttle）。

注册应用后，必须记下 Stack Exchange 提供的键。OAuth 步骤必然会用到**用户 ID** 和**客户端密钥**，在进行鉴权调用时，键是我们用来享受增加的访问次数的凭证。很重要的另一点是，正如文档中列出的，用户 ID 和键并不是保密的，而客户端密钥则是一个敏感信息，不应该被分发。

和前几章的做法一样，用环境变量配置应用的键。

```
$ export STACK_KEY="your-application-key"
```

通过程序与 API 交互前，我们需要用到的下一个工具是 API 客户端。这里我们主要使用名为 Py-StackExchange 的 Python Stack Exchange API 客户端。该客户端用一个直接的接口将 API 的端点与 Python 方法结合起来。

在我们的虚拟环境中，用以下命令从 CheeseShop 安装 Py-StackExchange。

```
$ pip install py-stackexchange
```

为了测试客户端并显示其核心的类，我们可以使用 Python 的交互式 shell 并观察 Stack Overflow 网站的一些通用统计数据。

```
>>> import os
>>> import json
>>> from stackexchange import Site
>>> from stackexchange import StackOverflow
>>> api_key = os.environ.get('STACK_KEY')
>>> so = Site(StackOverflow, api_key)
>>> info = so.info()
>>> print(info.total_questions)
10963403
>>> print(info.questions_per_minute)
2.78
>>> print(info.total_comments)
53184453
>>> print(json.dumps(info.json, indent=4))
{
  "total_votes": 75841653,
  "badges_per_minute": 4.25,
  "total_answers": 17891858,
  "total_questions": 10963403,
  "_params_": {
    "body": "false",
    "site": "stackoverflow.com",
    "comments": "false"
  },
  "api_revision": "2016.1.27.19039",
```

```
    "new_active_users": 35,
    "total_accepted": 6102860,
    "total_comments": 53184453,
    "answers_per_minute": 4.54,
    "total_users": 5126460,
    "total_badges": 16758151,
    "total_unanswered": 2940974,
    "questions_per_minute": 2.78
}
```

与 API 交互的核心类是 `stackexchange.Site`，可以用两个参数来实例化它：第一个是我们感兴趣的 Stack Exchange 网站的类定义（这里是 `stackexchange.StackOverflow`），第二个是可选的应用键。定义好 `Site` 对象后，可以用其方法访问 API。

在这个示例中，我们将用 `info()`方法访问`/info` 端点。返回的对象有一些属性。这个示例将打印 `total_questions`、`questions_per_minute` 和 `total_comments` 的值。可以在 `info.__dict__`字典中查看完整属性列表，此外，`info` 对象也有一个 `info.json` 属性，包含来自 API 的原始响应，而且把前面提到的所有属性都显示为 JSON 格式。

6.2.1 搜索带标签的问题

介绍完与 Stack Exchange API 的基本交互后，我们将用 Python 客户端来搜索问题。具体来说，我们将用话题标签作为关键词的过滤器。

以下代码执行搜索。

```
# Chap06/stack_search_keyword.py
import os
import json
from argparse import ArgumentParser
from stackexchange import Site
from stackexchange import StackOverflow
def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--tags')
  parser.add_argument('--n', type=int, default=20)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  my_key = os.environ.get('STACK_KEY')
  so = Site(StackOverflow, my_key)
  questions = so.questions(tagged=args.tags, pagesize=20)[:args.n]

  for i, item in enumerate(questions):
    print("{}) {} by {}".format(i,
                                item.title,
                                item.owner.display_name))
```

上述代码用 `ArgumentParser` 从命令行获取一些参数。`--tags` 选项用于传递我们搜索的标签列表，并以分号分隔。结果的数目由脚本中的`--n` 标记定义（默认是 20 个问题）。

可以用以下命令运行上述脚本。

```
$ python stack_search_keyword.py --tags "python;nosql" --n 10
```

标签列表外的双引号是必需的，因为分号是 Bash 这样的 shell 的命令分隔符，所以整行会分为两个独立的命令（第二个命令以 `nosql` 开头）。使用双引号后，`python;nosql` 字符串不会被当作单个字符串，整行是一个完整的命令。

标签列表通过 `questions()`方法的 `tagged` 参数传递到 API。该参数出现在一个布尔 AND 查询中，意味着检索的问题同时包含 `python` 和 `nosql` 标签。



布尔查询

`questions()`方法实现的是`/questions` 端点，该端点用 AND 连接多个标签。调用该方法的输出是一个包含这些标签的问题。

`search()`方式实现的是`/search` 端点，与上述 `questions()`方法相反的是，多个标签是用 OR 连接的。调用该方法的输出是一个包含任意给定标签的问题列表。

要注意的一个重要细节是，Py-StackExchange 客户端经常用懒惰列表来最小化底层 API 调用的次数。也就是说，`questions()`这样的方法返回的并不是一个列表，而是结果集合的一个包装器。对这个结果集合的直接迭代会尝试访问匹配查询的所有项（如果不小心处理，可能会违背使用懒惰列表的初衷），而不只是给定页面大小的列表。

因此，我们用 `slice` 运算符来显式调用特定数量的结果（通过命令行参数`--n` 指定，因此可以用 `args.n` 访问）。



Python 列表的切片和取值

`slice` 运算符是一个非常强大的工具，但其语法对 Python 的初学者来说有些令人困惑。以下示例总结了其主要用途。

```
# 选定从列表开始到末尾索引-1 的项
array[start:end]
# 选定列表所有项
array[start:]
# 选定从列表开始到末尾索引-1 的项
array[:end]
# 获得整个列表的一个副本
array[:]
# 按照步长值从列表中挑选项
array[start:end:step]
# 获得最后一项
array[-1]
#获得最后 n 项
array[-n:]
```

```
# 获得除最后 n 项外的所有项
array[:-n]
```

例如，思考以下示例。



```
>>> data = ['one', 'two', 'three', 'four', 'five']
>>> data[2:4]
['three', 'four']
>>> data[2:]
['three', 'four', 'five']
>>> data[:4]
['one', 'two', 'three', 'four']
>>> newdata = data[:]
>>> newdata
['one', 'two', 'three', 'four', 'five']
>>> data[1:5:2]
['two', 'four']
>>> data[-1]
'five'
>>> data[-1:]
['five']
>>> data[:-2]
['one', 'two', 'three']
```

值得注意的是，索引是从 0 开始的，因此 data[0]是第一项，data[1]是第二项，以此类推。

stack_search_keyword.py 脚本的输出结果是与 Python 和 NoSQL 相关的 10 个问题组成的一个序列，格式如下。

```
n) Question title by User
```

（通过 item 变量在 for 循环获得的）每个问题以及 title 或 tags 等属性被包装进 stackexchange.model.Question 模型类。通过 owner 属性将每个问题与提出问题的用户的模型链接起来。在这个示例中，我们获得了问题的 title 和用户的 display_name。

结果集合由最后的活动日期排序，也就是说，活跃时间越近（如有新的答案）则显示的排序越靠前。

排序效果受 questions()方法的 sort 属性影响，它可以采用以下值。

- activity（默认值）：靠前展示具有最新活动的问题
- creation：靠前展示最新创建的问题
- votes：靠前展示具有最高投票分数的问题
- hot：用公式计算出**热门问题**
- week：用公式计算出**本周问题**
- month：用公式计算出**月度问题**

6.2.2　搜索用户

介绍完如何搜索问题后，本节将介绍如何搜索特定的用户。过程与搜索问题的方法类似。以下示例建立在前面的示例基础上，这里扩展 ArgumentParser 来个性化设置一些搜索选项。

```
# Chap06/stack_search_user.py
import os
import json
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from stackexchange import Site
from stackexchange import StackOverflow
```

以下函数用于验证通过 ArgumentParser 传递的参数是否合法，如果参数非法，这些函数会抛出异常来阻止代码继续执行。

```
def check_sort_value(value):
  valid_sort_values = [
    'reputation',
    'creation',
    'name',
    'modified'
  ]
  if value not in valid_sort_values:
     raise ArgumentTypeError("{} is an invalid sort value".format(value))
  return value

def check_order_value(value):
  valid_order_values = ['asc', 'desc']
  if value not in valid_order_values:
    raise ArgumentTypeError("{} is an invalid order value".format(value))
  return value
```

接下来，check_sort_value() 和 check_order_value() 函数就可以作为相关参数的 type 在 ArgumentParser 定义中使用。

```
def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--name')
  parser.add_argument('--sort',
                      default='reputation',
                      type=check_sort_value)
  parser.add_argument('--order',
                      default='desc',
                      type=check_order_value)
  parser.add_argument('--n', type=int, default=20)
  return parser
```

实现用户搜索的这段代码的逻辑非常直接。

```
if __name__ == '__main__':
  parser = get_parser()
```

```
args = parser.parse_args()
my_key = os.environ.get('STACK_KEY')
so = Site(StackOverflow, my_key)

users = so.users(inname=args.name,
                 sort=args.sort,
                 order=args.order)
users = users[:args.n]

for i, user in enumerate(users):
  print("{}) {}, reputation {}, joined {}".format(i,
        user.display_name,
        user.reputation,
        user.creation_date))
```

stack_search_user.py 脚本照常用 ArgumentParser 来捕获命令行参数。与前面示例的不同之处在于对某些选项（如--sort 和--order）自定义数据类型。在深入介绍 ArgumentParser 数据类型前，我们先介绍一个用例。

假设我们想要搜索 Stack Exchange 的创始人（Jeff Atwood 和 Joel Spolsky）。查询语句如下所示。

```
$ python stack_search_user.py --name joel
```

该命令返回的输出结果与以下示例类似。

```
0) Joel Coehoorn, reputation 231567, joined 2008-08-26 14:24:14
1) Joel Etherton, reputation 27947, joined 2010-01-14 15:39:24
2) Joel Martinez, reputation 26381, joined 2008-09-09 14:41:43
3) Joel Spolsky, reputation 25248, joined 2008-07-31 15:22:31
# (snip)
```

实现/users 端点的 Site.users()方法的默认做法是返回一个用户列表，该列表的排列顺序依据声望值从大到小排列。

有趣的是，Stack Overflow 的创始人之一并不是用户列表中声望值最高的人。我们想要查询他是否为注册时间最早的人。这可以通过加入一些参数实现，如下所示。

```
$ python stack_search_user.py --name joel --sort creation --order asc
```

不出所料，现在输出结果不同了，Joel Spolsky 的注册时间最早。

```
0) Joel Spolsky, reputation 25248, joined 2008-07-31 15:22:31
1) Joel Lucsy, reputation 6003, joined 2008-08-07 13:58:41
2) Joel Meador, reputation 1923, joined 2008-08-19 17:34:45
3) JoelB, reputation 84, joined 2008-08-24 00:05:44
4) Joel Coehoorn, reputation 231567, joined 2008-08-26 14:24:14
# (snip)
```

这个示例中的 ArgumentParser 充分利用了可以为每个参数自定义数据类型的便利之处。

通常来说，传递给 `add_argument()` 函数的 `type` 参数使用内置的数据类型（如 `int` 或 `bool`），但在示例中，我们可以进一步扩展其功能来实现一些基本的数据验证功能。

问题是，传递给 `/users` 端点的 `sort` 和 `order` 参数仅接受有限的字符串集合。`check_sort_value()` 和 `check_order_value()` 函数只确认给定的值是否在接受的列表值中。如果没有出现，该函数会抛出 `ArgumentTypeError` 异常，`ArgumentParser` 会捕获该异常并向用户显示一条错误消息。例如，使用以下命令。

```
$ python stack_search_user.py --name joel --sort FOOBAR --order asc
```

使用 `--sort` 参数将产生如下输出。

```
usage: stack_search_user.py [-h] [--name NAME] [--sort SORT] [--order ORDER]
  [--n N]
stack_search_user.py: error: argument --sort: FOOBAR is an invalid sort value
```

如果没有实现自定义的数据类型，用错误的参数调用该脚本将导致 API 返回错误，Py-StackExchange 客户端会捕获该错误，并显示 `StackExchangeError` 及相关的错误信息。

```
Traceback (most recent call last):
  File "stack_search_user.py", line 48, in <module>
# (long traceback)
stackexchange.core.StackExchangeError: 400 [bad_parameter]: sort
```

换句话说，自定义数据类型允许我们处理错误，并向用户显示一个更漂亮的消息。

6.3　处理 Stack Exchange 的存档数据

Stack Exchange 网络还提供了完整的存档数据，可以通过网络存档下载。数据格式是有很高压缩比的 7Z。为了读取和抽取这种格式的文件，Windows 系统需要下载 7-zip 程序，Linux/Unix 和 macOS 系统也需要下载相应版本。

在撰写本书时，Stack Overflow 的存档是用多个独立的压缩文件提供的，每个压缩文件表示数据集的一个实体或表格。例如，stackoverflow.com-Posts.7z 文件包含帖子表格（即问题和回答）的存档。该文件的第一版于 2016 年发布，数据大小约有 7.9GB，压缩前的大小为 39GB（约是压缩后的 5 倍）。Stack Exchange 其他网站的数据集更小一些，因此它们为网站的表格提供单个压缩文件包。

例如，电影和电视爱好者在 Movies & TV 网站上提问和回答有关电影和电视主题的问题，可以通过单个 movies.stackexchange.com.7z 文件下载来自该网站的数据。解压该文件后可以看到 8 个文件（对应 8 个实体），其内容信息如表 6-1 所示。

表 6-1 Movies & TV 的文件

文件名称	表示实体
movies.stackexchange.com	勋章
movies.stackexchange_1.com	评论
movies.stackexchange_2.com	发帖历史
movies.stackexchange_3.com	帖子链接
movies.stackexchange_4.com	帖子（问题和回答）
movies.stackexchange_5.com	标签
movies.stackexchange_6.com	用户
movies.stackexchange_7.com	投票

不同网站的不同数据存档遵循相同的命名规则。例如，对每个网站来说，website_4.com 文件名包含的都应该是帖子列表。

存档文件的内容都是以 XML 格式表示的，其结构通常如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
  <entity>
    <row [attributes] />
    <!-- more rows -->
  </entity>
```

第一行是标准的 XML 声明。这个 XML 文档的根元素是一个实体类型（如 posts、users 等）。特定实体的每一项用<row>元素表示，而它又会包含多个属性。例如，一小段 posts 实体的片段如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
  <posts>
    <row Id="1" PostTypeId="1" ... />
    <row Id="2" PostTypeId="2" ... />
    <row Id="3" PostTypeId="1" ... />
    <!-- more rows -->
  </posts>
```

因为 posts 元素用于表示问题和回答，所以每一行的属性可以不相同。

表 6-2 列出了用于表示一个问题的<row>元素最重要的一些属性。

表 6-2 表示问题的<row>元素的属性

属性名称	描　述
Id	每个帖子的唯一标识符
PostTypeId	对于问题，它的值就是 1
AcceptedAnswerId	标记为已采纳回答的帖子的 ID
CreationDate	ISO 8601 格式的问题发布日期

6

（续）

属性名称	描　　述
Score	问题的支持票数减去不支持票数的值
ViewCount	问题被查看的次数
Title	问题的题目
Body	问题的完整文本
OwnerUserId	发布问题的用户的 ID
Tags	问题的标签列表
AnswerCount	问题的回答数量
CommentCount	问题的评论数量
FavoriteCount	将问题标记为喜欢的用户数量

类似地，表 6-3 列出了用于表示一个回答的<row>元素的主要属性。

表 6-3　表示回答的<row>元素的属性

属性名称	描　　述
Id	每个帖子的唯一标识符
PostTypeId	对于回答，它的值是 2
ParentId	回答所对应问题的 ID
OwnerUserId	发布回答的用户的 ID
Score	回答的支持票数减去不支持票数的值
CommentCount	回答的评论数量
Body	回答的完整文本

本书主要处理 JSON Lines 格式的数据，因为 JSON Lines 格式可以提供更方便的数据表示。

以下的 stack_xml2json.py 脚本用于将存档数据的 XML 格式转换为我们更熟悉的 JSON Lines 格式。这里也进行了一些基本清洗。

虽然 Python 的标准库对 XML 有很好的支持，但还可以考虑一些有趣的包，如 lxml。这个库提供了 Python 库和低级 C 语言库（libxml2 和 libxslt）的结合。它结合了两个库极致的性能和完整的特征，并保持了 Python 库的良好设计和易于使用的特性。

潜在的坏处在于安装过程，可能会发生 libxml2 和 libxslt 库不兼容的情况，因为它们都需要各自的依赖。该安装过程也是使用 pip，正如我们前面安装所有包一样。

```
$ pip install lxml
```

文档（http://lxml.de/installation.html）中详细介绍了 C 语言库的优先选择版本，以及一些优化。根据平台和系统配置的不同，一些细节差异可能会导致安装发生错误。完整的故障排除方案并不在本书的范围内，但网上提供了解决各种故障的很多材料（Stack Overflow 本身就是非常优

质的信息源）。

好在 lxml 被设计成与 ElementTree 包（Python 标准库的一部分）兼容，因此，安装 lxml 对本节来说并不是特别重要。这里我们建议尽量尝试安装，如果情况变得很复杂，那么就退回来，不用过多担心。实际上，如果 lxml 缺失，以下脚本可以退回以使用 ElementTree。

```
# Chap06/stack_xml2json.py
import json
from argparse import ArgumentParser
from bs4 import BeautifulSoup
try:
  from lxml import etree
except ImportError:
  # 如果未安装 lxml，退回使用 ElementTree
  import xml.etree.ElementTree as etree

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--xml')
  parser.add_argument('--json')
  parser.add_argument('--clean-post',
                      default=False,
                      action='store_true')
  return parser

def clean_post(doc):
  try:
    doc['Tags'] = doc['Tags'].replace('><', ' ')
    doc['Tags'] = doc['Tags'].replace('<', '')
    doc['Tags'] = doc['Tags'].replace('>', '')
  except KeyError:
    pass
  soup = BeautifulSoup(doc['Body'], 'html.parser')
  doc['Body'] = soup.get_text(" ", strip=True)
  return doc

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  xmldoc = etree.parse(args.xml)
  entity = xmldoc.getroot()
  with open(args.json, 'w') as fout:
    for row in entity:
      doc = dict(row.attrib)
      if args.clean_post:
        doc = clean_post(doc)
      fout.write("{}\n".format(json.dumps(doc)))
```

对于 lxml 非常重要的一点是，加入 `try/except` 模块来捕获 `ImportError`。当尝试导入一个不在 Python 路径中的包/模块时，就会发生这种异常。

stack_xml2json.py 脚本再次用 ArgumentParser 来捕获命令行参数。这段代码使用了一个可选的--clean-post 标记，因此我们可以区别 posts 实体和其他实体，其中 posts 实体需要一些特殊的处理，我们稍后会介绍这一点。

例如，将 Movies & TV 数据集中的标签文件转换为 JSON，可以使用以下命令。

```
$ python stack_xml2json.py --xml movies.stackexchange_5.com --json
  movies.tags.jsonl
```

以上命令会生成 movies.tags.jsonl 文件，其中每个标签都表示为在一行中的 JSON 文档。我们用 Bash 提示符简单地审查该文件。

```
$ wc -l movies.tags.jsonl
```

以上命令计算文件中的行数，即标签的数量。

```
2218 movies.tags.jsonl
```

如果想要审查第一个 JSON 文档，即文件的第一行，使用以下命令。

```
$ head -1 movies.tags.jsonl
```

输出结果如下所示。

```
{"Count": "108", "WikiPostId": "279", "Id": "1", "TagName": "comedy",
"ExcerptPostId": "280"}
```

该脚本用 lxml.etree 解析 XML 文档，并放入 xmldoc 变量，该变量中存放文档的一种树结构表示，使用的是 ElementTree 实例。为了获取树中的特定对象，我们使用 getroot()方法作为项的节点，从该节点开始迭代每一行。该方法返回的是一个 Element 对象，存放在 posts 变量中。正如前面描述的，数据存档中的每个文件都有同样的结构，其中根元素是实例名称（如 <tags>、<posts>等）。

为了获取给定实体的单独的项（即对应的行），可以用一个常规的 for 循环从根元素开始迭代。行也是 Element 类的实例。从 XML 转换到 JSON 需要将元素属性转存到一个字典中。这一步是通过简单地将 row.attrib 对象映射到一个 dict 实现的，这样可以序列化 JSON。然后可以再通过 json.dumps()将该字典转存到输出文件中。

正如前面提到的，posts 实体需要特殊的处理。当用这段代码将帖子转换成 JSON 后，我们还需要额外的--clean-post 参数。

```
$ python stack_xml2json.py --xml movies.stackexchange_4.com --json
  movies.posts.jsonl --clean-post
```

该命令产生 movies.posts.jsonl 文件。额外的标记会调用 clean_post()函数，对包含单个帖子的字典进行数据清洗。具体来说，需要清洗的属性是 Tags（针对问题）和 Body（针对问题和回答）。

Tags 属性是<tag1><tag2>...<tagN>格式中的一个字符串，这里的尖括号用于分隔每个标签。通过去除每个标签的括号并加上空格，我们可以简化之后检索标签名称的过程。例如，一个电影问题可以被标记为喜剧和浪漫剧。在清洗前，这里 Tags 属性的值就是<comedy><romance>，清洗后，标签会转换为 comedy romance。转换的代码包含在一个捕获 KeyError 的 try/except 模块中；当 Tags 键没有出现在文档中时，它就会抛出异常，也就是说，当个我们处理回答时就会抛出异常（因为只有问题有标签）。

接下来的步骤是用 Beautiful Soup 抽取文本。实际上，XML 中偶尔会包含一些 HTML 代码，这些代码是关于段落格式的，我们并不需要用这些信息来分析问题的文本内容。Beautiful Soup 中的 get_text()函数会去除这些 HTML 代码，并返回我们需要的文本。

6.4　问题标签的文本分类

本节是关于监督学习的。我们把向问题分配标签定义为文本分类问题，并将文本分类问题应用于来自 Stack Exchange 的问题数据集。

在详细介绍文本分类前，我们首先思考一下来自 Movies & TV Stack Exchange 网站的以下问题（问题的标题和正文部分被合并了）。

> “在电视剧《豪斯医生》的哪一集里豪斯医生雇了一个女人通过假死欺骗团队？我记得有一个（据说已经死亡的）女人醒来并且和豪斯医生击掌庆祝。是哪一集来着？”

以上问题询问了热播电视剧《豪斯医生》特定剧集的详细信息。正如前面介绍的，Stack Exchange 上的问题都会被打上标签，这样可以快速地识别问题的主题。用户给这个问题打的标签是 house 和 identify-this-episode：第一个标签指明了电视剧本身，第二个标签描述了问题的性质。有些人可能认为只用一个标签（这里使用 house）来标记问题就已经足够描述其主题。与此同时，我们可以看到，这两个标签并不是相互排斥的，因此多个标签有助于更好地表达问题。

虽然出于更好地理解文档的主题而为文档分配标签看起来非常直观，但这确实是一个将项（真实对象、人、国家、概念等）分配到类的长期存在的问题。作为研究的一个领域，分类涉及多个学科，从哲学到计算机科学，再到商业管理。在本书中，分类被视为一个机器学习问题。

6.4.1　监督学习和文本分类

监督学习是从打标签的训练数据中推断函数的一个机器学习领域。作为监督学习的一个特定类型，分类的用途是基于训练数据以及训练数据属于哪一类，将新的项分配到正确的类。

分类的实际示例如下所示。

- **邮件过滤**：确定一封新的电子邮件是否为垃圾邮件

- **语言识别**：自动检测文本的语言
- **类型识别**：自动检测文本的类型/主题

这些示例描述了分类的不同变体。

- 垃圾邮件过滤是一个二元分类的任务，因为只有两个可能的类：是垃圾邮件或者不是垃圾邮件。这也是一个一元标签分类问题，因为只能用一个标签来描述给定的文档。
- 语言检测可以用不同的方法实现。这里假设一篇文档只使用一种语言，那么语言识别也是一个一元标签分类问题。就类的数量来说，也可以将其看作一个二元分类问题（例如，文档是否用英文撰写），但对其最好的描述可能还是多元分类问题，即类的数量大于 2，而且文档可以分配到任意一类。
- 最后，类型识别（或主题分类）是一个典型的多元分类和多标签问题：可能的类的数量大于 2，并且同一篇文档可以分配到多个类别（正如《豪斯医生》的问题示例）。

作为监督学习的特定类型，分类需要训练数据，即带有正确分类标签的数据实例。一个分类器在学习（或训练）和预测这两个阶段运行，如图 6-4 所示。

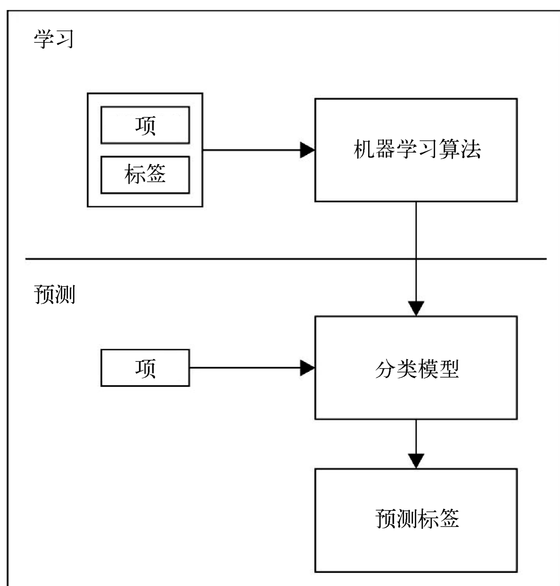


图 6-4　监督学习的通用框架

在学习阶段，将训练数据提供给机器学习算法。因此，算法的输入是部分字段的`(item, labels)`对，其中给定项的正确标签是已知的。这个过程构建了分类器模型，接下来该模型为新的数据预测标签。在这个阶段，输入是一个新的项，输出是通过算法分配给该项的标签。

在构建分类模型的过程中，机器学习算法**关注**一些元素，以学习如何分类文档。

在机器学习中，表示文档（更一般地说是对象）的一种常见方式是用一个 n 维向量，向量的每一个元素称作**特征**。从原始数据构建此类向量的过程通常尽量表现更多的信息，该过程也称作**特征抽取**或**特征工程**（当这个步骤中涉及专家知识时，多使用后者）。

从实用的角度来看，选择正确的特征并用正确的方式来表示它们能极大提升机器学习算法的性能。通常情况下，可以用一组很明显的特征来获得较好的性能。通过直觉来测试不同方法的迭代**试错**（trial-and-error）过程经常出现在开发的早期阶段。在文本分析的情境下，一开始通常采用词袋方法，即考虑单词的频率。

为了清楚从文档到向量的过程，我们来思考表 6-4 所示的示例。这里，两个文档的示例语料以原始和向量化的方式展现。

表 6-4 从文档到向量

原始文档	
Doc1	John is a programmer.
Doc2	John likes coding. He also likes Python.
向量化文档	
Doc1	[1, 1, 1, 1, 0, 0, 0, 0, 0]
Doc2	[1, 0, 0, 0, 2, 1, 1, 1, 1]
特征	[John, is, a, programmer, likes, coding, he, also, Python]

在以上示例中，每个文档的向量化版本包含 9 个元素。语料中不同单词的数量表示向量的维度数。虽然这里并没有考虑单词在原始文档中出现的语法顺序，但向量中元素的顺序非常重要。例如，每个向量中的第一个元素表示单词 John、第二个元素表示 is、第五个元素表示 likes，等等。

这里的每个单词用其原始频率表示。其他常用的方法还包括二元表示（所有非零的次数设置为 1，仅仅表示该单词在文档中出现了），以及一些更复杂的统计方法，如 TF-IDF。

从更高的视角来看，从文档创建向量的任务非常直接，但需要一些非常琐碎但重要的操作，包括对文本的词项化和归一化。

好在 scikit-learn 提供了一些帮助生成文档向量的易用工具。`CountVectorizer` 和 `TfidfVectorizer` 类就是我们需要的实用工具。它们都属于 `feature_extraction.text` 子包，因此可以用以下代码导入。

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
```

具体来说，`CountVectorizer` 计算的是原始频率和二元表示（`binary` 属性默认设置为 `False`），而 `TfidfVectorizer` 会将原始的频率转换为 TF、TF-IDF 或归一化的 TF-IDF。向量的结果都会随着可能传入构造器的一些参数而改变。

以下是一个使用 `TfidfVectorizer` 类的示例。

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
... "Peter is a programmer.",
... "Peter likes coding. He also likes Python"
... ]
>>> vectorizer = TfidfVectorizer()
>>> vectors = vectorizer.fit_transform(corpus)
>>> vectors
<2x8 sparse matrix of type '<class 'numpy.float64'>'
  with 9 stored elements in Compressed Sparse Row format>
>>> vectors[0]
<1x8 sparse matrix of type '<class 'numpy.float64'>'
  with 3 stored elements in Compressed Sparse Row format>
>>> vectors[1]
<1x8 sparse matrix of type '<class 'numpy.float64'>'
  with 6 stored elements in Compressed Sparse Row format>
>>> print(vectors)
  (0, 6)    0.631667201738
  (0, 3)    0.631667201738
  (0, 5)    0.449436416524
  (1, 7)    0.342871259411
  (1, 0)    0.342871259411
  (1, 2)    0.342871259411
  (1, 1)    0.342871259411
  (1, 4)    0.685742518822
  (1, 5)    0.243955725
>>> vectorizer.get_feature_names()
['also', 'coding', 'he', 'is', 'likes', 'peter', 'programmer', 'python']
```

可以观察到，正如我们所预料的那样，向量化操作默认执行分词，但单词 a 并没有出现在特征列表中（实际上，向量是 8 维的，而不是我们前面介绍的 9 维）。

这是因为默认的词项生成过程用一个正则表达式来捕获所有大于等于 2 个字母符号的词项，并将标点符号和空格作为分隔符。默认情况下，词项会归一化为小写形式。

以下列举了一些会影响 `TfidfVectorizer` 的有趣属性。

- `tokenizer`：可调用来重写分词步骤
- `ngram_range`：一个元组(`min_n, max_n`)，可用来设置 n-gram 作为词项，而不是用单个单词作为词项
- `stop_words`：一个明确的停用词列表（默认不执行停用词移除步骤）
- `lowercase`：一个默认为 `True` 的布尔值
- `min_df`：定义最小的文档频率阈值。它可以是一个整型值（例如，5 表示词项在文档中出现了 5 次及 5 次以上）或者用一个范围在区间[0.0, 1.0]的浮点值来表示文档的比值。它默认情况下设置为 1，即没有阈值
- `max_df`：定义最大的文档频率阈值。与 `min_df` 类似，它可以是整型或者浮点型。它默认情况下设置为 1.0（即文档的 100%），意味着没有阈值

- ❑ `binary`：一个默认为 `False` 的布尔值。当它设置为 `True` 时，所有非零项的次数都设置为 1，意思是 TF-IDF 的 TF 成分是二元值，但最终的取值仍然受 IDF 值影响
- ❑ `use_idf`：一个默认为 `True` 的布尔值。当它设置为 `False` 时，IDF 权重会被禁用
- ❑ `smooth_idf`：一个默认为 `True` 的布尔值。对文档频率启用+1 平滑可以防止除零错误发生
- ❑ `sublinear_tf`：一个默认为 `False` 的布尔值。当它设置为 `True` 时，使用次线性 TF 值缩放，即用 1+log(TF)来替换 TF 值

下一节介绍了三个常用的分类方法，这些方法都是集成在 scikit-learn 中的。

6.4.2　分类算法

本节将简要列出三种用于文本分类的常见机器学习算法：**朴素贝叶斯**（naive Bayes，NB）、**K 最邻近**（k-nearest neighbor，k-NN)和**支持向量机**（support vector machine，SVM）。本节的目的不是深入探讨这些算法的细节，而是简单地介绍 scikit-learn 中可以直接调用的模块。查看 scikit-learn 的文档很有价值，因为 scikit-learn 提供了丰富的算法（监督学习算法：http://scikit-learn.org/stable/supervised_learning.html）。

值得注意的是，不同算法对于不同数据会呈现不同的表现，因此，在开发分类系统的过程中，我强烈建议你尝试不同的算法实现。scikit-learn 算法提供的接口非常简单易用，只修改一两行代码就能更换算法。

1. 朴素贝叶斯

我们首先介绍朴素贝叶斯算法，它是基于贝叶斯定理的分类器家族中的一员，假设特征间是独立的（因此称为朴素）。

贝叶斯定理（也称作贝叶斯定律或贝叶斯规则）描述的是一个事件的条件概率。

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

在我们已经观察到事件 B 发生的前提下，等式左边的项是观察到事件 A 的概率。

为了将等式应用于具体场景，我们思考一下如下过滤垃圾邮件的示例。

$$P(\text{Spam} \mid \text{“money”}) = \frac{P(\text{“money”} \mid \text{Spam})P(\text{Spam})}{P(\text{“money”})}$$

该等式展示了如何计算将 Spam 标签分配给一个包含单词 money 的新文档的概率。在条件概率下，该问题通常描述为：在观察到单词 money 的前提下，为这篇文档分配 Spam 标签的概率是多少？

通过使用训练数据，可以在训练步骤中计算出等式右边的三个概率。

由于分类器并不使用单个单词/特征为文档分配特定的类，我们来思考以下更常见的情况。

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)} \propto P(c)\prod_{t \in d} P(t \mid c)$$

上述等式描述了为给定文档 d 分配类 c 的概率。根据独立性假设，该文档被描述为一组独立的项 t 构成的序列。由于特征相互独立，概率 $P(d|c)$可以写成文档 d 中每个项 t 的 $P(t|c)$的乘积。

细心的你可能已经意识到，上述等式的最后一部分并没有分母，其中我们将文档表示为独立项的序列。这是一个简单的优化，因为 $P(d)$对不同的取值来说都是一样的，并不需要为了计算和比较各个类的分值而计算处理。因此，等式的最后一部分使用了一个**成正比符号**，而不是**等号**。

这就是我们介绍的全部数学背景知识，毕竟本书并不想专注于算法的数学部分。如果感兴趣，你可以自行深入探究（参考 Sebastian Raschka 撰写的《Python 机器学习》）。

在 scikit-learn 中，`naive_bayes` 子包提供了对朴素贝叶斯算法的支持。例如，多项式朴素贝叶斯分类器是文本分类中最常用的分类器，由 `MultinomialNB` 类实现。

```
from sklearn.naive_bayes import MultinomialNB
```

2. K 最邻近

k-NN 算法也是用于分类和回归的方法。该算法的输入包含特征空间中最近的 k 个训练示例，即 k 个最相似的文档。算法的输出是通过近邻的大多数投票获得的对特定文档的分类标签。

在训练阶段，训练集中文档的多维向量表示与它们的标签一同存放。在预测阶段，新的文档会查询与自身最类似的 k 个文档，这里的 k 是用户自定义的常数。这 k 个文档（最邻近）中最多的类就会被分配给新的文档。

k-NN 是一种非参数化方法，因为该算法由数据（即文档向量）确定类型，而不是由对数据估计的参数确定（对朴素贝叶斯来说，我们用估计的概率来确定类型）。

算法背后的直觉非常直接，但还有两个开放问题。

- 如何计算向量间的距离？
- 如何选择最佳的 k 值？

第一个问题有多个有效的答案。最常见的选项是欧氏距离。scikit-learn 中默认实现的是闵式距离，在 scikit-learn 中使用预定义参数时，它就等同于欧氏距离。

第二个问题的答案有些复杂，取决于具体的数据。选择较小的 k 意味着该算法会更多地受噪

声影响。选择较大的 *k* 值会使得算法的计算复杂度变大。一种常见的做法是选择特征数量 *n* 的平方根，即 `k = sqrt(n)`，但需要尝试。

在 scikit-learn 中，`neighbors` 子包提供了对 k-NN 算法的支持，如下所示。

```
from sklearn.neighbors import KneighborsClassifier
```

`KneighborsClassifier` 类接收一些参数，其中包括 `k` 和 `distance`。想详细了解不同距离度量的可用选项，可以查看 `sklearn.neighbors.DistanceMetric` 类中的文档。

3. 支持向量机

SVM 是用于分类和回归的监督学习模型。

SVM 应用的是二元分类方法，也就是说，新的文档会分配到两个类的其中之一。该方法可以通过一对多或一对一方法扩展到多元分类。一对多方法给定的类会与其他所有类进行比较，而一对一方法会考虑到所有可能的分类。在一对多的方法中，构建 *n* 个分类器，这里 *n* 就是类的总数。具有最大输出值的类就是分配给文档的类。在一对一的方法中，每个分类器为选定的类增加一个计数器。最后，拥有最多投票的类就是分配给文档的类。

SVM 系列的分类器因高效的文本分类效果而知名，而文本分类任务中的数据通常用高维空间表示。当数据的维数大于示例数据的数量时，分类器仍然有效。但如果特征的数量远大于示例数据的数量（即训练集中只有较少文本），则该方法会丧失有效性。

正如其他算法一样，SVM 算法在 scikit-learn 中也有实现（通过 `svm` 子包），具体的导入方法如下所示。

```
from sklearn.svm import LinearSVC
```

`LinearSVC` 类实现一个线性核，也就是其中一个可用的核函数。简单来说，核函数可以看作计算示例数据对之间相似度的相似函数。scikit-learn 库提供了常用核，也提供了定制化实现。`SVC` 类可以通过 `kernel='linear'` 参数实例化。虽然该分类器也是基于一个线性核，但底层的实现不同：这里 `SVC` 是基于 SVM 的常用库 `libSVM` 的，而 `LinearSVC` 是基于 `liblinear` 的，`liblinear` 是一个更有效的实现，并针对线性核做了优化。

6.4.3　评估

讨论了不同的分类算法后，本节将尝试解决一个有趣的问题：如何选择**最佳**算法？

分类方法用四种类别来定义分类器的结果：

- 该文档属于类 *C*，且系统将其分配给类 *C*（真阳性，True Positive，TP）
- 该文档不属于类 *C*，但系统将其分配给类 *C*（假阳性，False Positive，FP）

- 该文档属于类 C，但系统并未将其分配给类 C（假阴性，False Negative，FN）
- 该文档不属于类 C，且系统并未将其分配给类 C（真阴性，True Negative，TN）

通常用名为混淆矩阵的表格表示这些结果。对于二元分类问题，混淆矩阵如表 6-5 所示。

表 6-5　混淆矩阵

	预测类：C	预测类：不是 C
正确类：C	TP	FN
正确类：不是 C	FP	TN

混淆矩阵可以扩展，以用于多元分类的问题。一旦获得测试集中所有样例数据的结果，就可以计算不同的性能指标了，如下所示。

- **准确率**：分类器正确的概率是多少？（TP+TN）/总数
- **误分类率**：分类器错误的概率是多少？（FP+FN）/总数
- **精确率**：在分配到类 C 的数据中，分类器准确的概率是多少？TP/预测到类 C 的数量
- **召回率**：类 C 的文档中被准确识别的比率是多少？TP/类 C 的数量

这些指标都提供一个在 0~1 的值。准确率和召回率通常会合并为名为调和平均（也称作 F 值或 F1）的单个值。

在多元分类场景中，不同类的 F 值可以通过多种方式进行平均。典型的方法称作 micro-F1，其中每个决策都具有相同的平均权重。在另一种方法 macro-F1 中，每个类都具有相同的平均权重（参见 Fabrizio Sebastiani 撰写的 *Machine Learning in Automated Text Categorization*）。

到目前为止，我们在假设已有分割好的训练集和测试集（运行评估的数据集）的前提下介绍了评估方法。一些非常著名的数据集清晰标注了训练集/测试集的分割，因此研究者可以重现结果。有时这种分割并不明显，因此需要预留一部分标注数据用于测试。较小的测试集意味着评估可能不太准确，而较大的数据集意味着训练阶段的数据较少。

一种可行的解决方案叫作**交叉检验**，其中最常用的方法是 ***k* 折检验**。其基本思想是每次用不同的训练/测试分割来多次重复评估过程，然后取一个聚合值（如平均值）。

例如，我们可以用 10 折检验将数据集分割成 10 个不同的子集。我们将不断选择其中的一个折作为测试集，并将剩余的折合并为一个训练集。在单个折上运行评估后，保存结果并进行下一次迭代。现在第二个折变成测试集，其他 9 个折变成训练集。在 10 次迭代后，对所有的评估值取平均。k 折检验常用的方法是 10 折（90/10 的训练集/测试集分割）和 5 折（80/20 的训练集/测试集分割）。使用交叉检验的主要优点是，即使对于小数据集，我们也有信心可以获得较高的准确率。

6.4.4 Stack Exchange 数据的文本分类

介绍完分类、相关算法和评估后，本节会将我们学到的方法应用于 Movies & TV 网站的问题数据集。

由于用标签名来表示类，我们将挑选数据集以避免过于稀少而难以捕捉的类别。我们设定阈值为 10，也就是说，忽略出现次数少于 10 的标签。该阈值可以任意设定，也可以拿其他数字来实验。此外，帖子文件既包含问题又包含答案，但我们只关注问题（可以从 `PostTypeId="1"` 属性看出来），因此这里丢弃所有的回答。

以下脚本读取包含标签和帖子的 JSON Lines 文件，并生成一个只带有标签的 JSON Lines 文件，且标签的频率满足我们的阈值要求。

```
# Chap06/stack_classification_prepare_dataset.py
import os
import json
from argparse import ArgumentParser

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--posts-file')
  parser.add_argument('--tags-file')
  parser.add_argument('--output')
  parser.add_argument('--min-df', type=int, default=10)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  valid_tags = []
  with open(args.tags_file, 'r') as f:
    for line in f:
      tag = json.loads(line)
      if int(tag['Count']) >= args.min_df:
        valid_tags.append(tag['TagName'])

  with open(args.posts_file, 'r') as fin, open(args.output, 'w') as fout:
    for line in fin:
      doc = json.loads(line)
      if doc['PostTypeId'] == '1':
        doc_tags = doc['Tags'].split(' ')
        tags_to_store = [tag for tag in doc_tags
                         if tag in valid_tags]
        if tags_to_store:
          doc['Tags'] = ' '.join(tags_to_store)
          fout.write("{}\n".format(json.dumps(doc)))
```

该脚本假设我们已经将标签和帖子文件从 XML 格式转换为了 JSON Lines 格式。

可以用以下命令执行这个脚本。

```
$ python stack_classification_prepare_dataset.py \
  --tags-file movies.tags.jsonl \
  --posts-file movies.posts.jsonl \
  --output movies.questions4classification.jsonl \
  --min-df 10
```

这个脚本输出的文件 movies.questions4classification.jsonl 只包含满足条件的标签对应的问题。我们可以看到它与原始的帖子文件的差别。

```
$ wc -l movies.posts.jsonl
  28146 movies.posts.jsonl
$ wc -l movies.questions4classification.jsonl
  8413 movies.questions4classification.jsonl
```

接下来对 Movies & TV 数据集进行分类。遵循前面讨论的内容，我们将使用一个 10 折交叉检验。也就是说，问题语料将分为 10 折，分类任务将用 90%的数据做训练，并用剩下的 10%做测试，以进行 10 次迭代，每次迭代都用不同折进行测试。

```
# Chap06/stack_classification_predict_tags.py
import json
from argparse import ArgumentParser
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import f1_score
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.cross_validation import cross_val_score
import numpy as np

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--questions')
  parser.add_argument('--max-df', default=1.0, type=float)
  parser.add_argument('--min-df', default=1, type=int)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  stop_list = stopwords.words('english')

  all_questions = []
  all_labels = []
  with open(args.questions, 'r') as f:
    for line in f:
      doc = json.loads(line)
      question = "{} {}".format(doc['Title'], doc['Body'])
      all_questions.append(question)
```

```
        all_labels.append(doc['Tags'].split(' '))

    vectorizer = TfidfVectorizer(min_df=args.min_df,
                                 stop_words=stop_list,
                                 max_df=args.max_df)
    X = vectorizer.fit_transform(all_questions)
    mlb = MultiLabelBinarizer()
    y = mlb.fit_transform(all_labels)
    classifier = OneVsRestClassifier(LinearSVC())

    scores = cross_val_score(classifier,
                             X,
                             y=y,
                             cv=10,
                             scoring='f1_micro')
    print("Average F1: {}".format(np.mean(scores)))
```

stack_classification_predict_tags.py 脚本通过 ArgumentParser 接收 3 个参数。第一个参数--questions 是必备的，用于指定包含问题的.jsonl 文件。另外两个参数--min-df 和--max-df 可以在构建特征向量时影响分类器的行为。这两个参数的默认值不影响特征抽取的过程，因为这两个值分别设定为 min_df=1 和 max_df=1.0，也就是说，所有特征（单词）都将被包括在内。

可以用以下命令执行这个脚本。

```
$ python stack_classification_predict_tags.py \
  --questions movies.questions4classification.jsonl
```

输出结果如下所示。

```
Average F1: 0.6271980062798452
```


我们来详细查看该脚本，从命令行解析参数后，用 NLTK 将一个停用词列表导入 stop_list 变量。这是一个常用英语停用词列表（约有 130 个单词，其中包括冠词、连词、代词等）。下一步是读取输入文件，将每个 JSON 文档导入内存，并创建一个输入到分类器的数据结构。此外，我们将在 all_questions 变量创建一个所有文档的列表，其中每个问题的标题和正文内容是拼接起来的。同时，我们将追踪 all_labels 变量中的原始标签（即分类器的标签/类）。

对于特征抽取步骤，我们将创建 TfidfVectorizer 的一个实例，将停用词列表、最小文档频率和最大文档频率作为构造器的参数。向量器将为分类器创建特征向量，从原始文本开始，将每个单词转化为相应的 TF_IDF 值。停用词不包括在向量中。类似地，不满足规定频率要求的单词也会被丢弃。

从原始文本到向量的转化通过向量器的 fit_transform()方法实现。

由于我们想要实现多标签，为了正确地将所有标签映射到二元向量，还需要一个额外的步骤。这是必需的，因为分类器期望的是一个二元向量列表作为输入，而不是一个类名称列表。为了理

解 MultiLabelBinarizer 如何实现转换，可以运行以下代码。

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> labels = [['house', 'drama'], ['star-wars'], ['drama', 'uk']]
>>> y = mlb.fit_transform(labels)
>>> y
array([[1, 1, 0, 0],
       [0, 0, 1, 0],
       [1, 0, 0, 1]])
>>> mlb.classes_
array(['drama', 'house', 'star-wars', 'uk'], dtype=object)
```

至此，语料和标签都转换成了分类器可以理解的格式。关于命名的注意事项：以大写 X 表示语料、小写 y 表示标签是机器学习中的常见惯例，机器学习教材中经常出现这一点。如果不喜欢单个字母的变量，也可以对其重新命名。

对于分类器，我们选择线性核的 SVM 分类器 LinearSVC。如果感兴趣，也可以尝试其他类型的分类器。由于该任务是一个多元标签分类任务，我们将选择一对多的方法，该方法由 OneVsRestClassifier 类实现，它将一个实际分类器实例作为第一个参数，并用其执行所有的分类任务。

最后一个步骤就是分类。正如前面介绍的，我们希望做交叉检验，不是直接使用分类器对象，而是将它与数据向量、标签和一些其他参数一同传递给 cross_val_score() 函数。该函数的目的是根据 cv=10 参数（意味着使用 10 折）迭代数据、执行分类，并最后显示评估值（本例使用的是微平均的 F1 值）。

交叉检验函数的返回值是一个 NumPy 数组，每个折对应一个值。通过使用 numpy.mean() 函数，我们可以聚合这些值并显示它们的算术平均值。

值得注意的是，最后的 F1 值约为 0.62，不好也不坏。它告诉我们分类系统还远远不够完美，但截至目前，尚不明确性能是否还有提升的空间，以及如何提升。这里的评估分数值并没有告诉我们分类器的具体表现如何，或者是否任务太难以至于无法达到较好的性能。聚合的评估值是比较不同分类器性能的一种方法，也能够评估同一个分类器在设置不同参数时的性能。

我们可以指定一个额外的参数来过滤特别稀少的特征，并重新运行代码。

```
$ python stack_classification_predict_tags.py \
  --questions movies.questions4classification.jsonl \
  --min-df 5
```

通过设置最小文档频率为可以将特征频率的长尾分布截断的 5，我们可以得到以下输出。

```
Average F1: 0.635184845312832
```

这个示例表明，对配置进行简单的调整可以得到不同的结果（好在这里得到了更好的结果）。

6.4.5 在实时应用中嵌入分类器

前一节通过学术思维介绍了如何解决分类问题：从一组预先标记的文档开始，运行一批实验，执行交叉检验以获得评估度量值。如果我们想用不同的分类器来实验、调整特征抽取过程，并理解哪个分类器以及哪个配置对于给定的数据集效果最佳，那么这肯定是一个有趣的过程。

本节将进一步讨论将分类器（更一般地说是一个机器学习模型）嵌入应用的一些简单步骤，嵌入后的分类器可以与用户实时交互。这种应用的常见示例包括搜索引擎、推荐系统、垃圾邮件过滤器，以及日常生活中可能用到的很多其他智能应用，但平时我们可能意识不到这些应用使用了机器学习技术。

本节创建的应用将执行以下步骤。

- 训练分类器（学习步骤，线下运行）
- 将分类器保存在磁盘上，以便可以从实时应用中导入
- 将分类器嵌入应用，使其可以与用户实时交互（预测步骤，实时运行）

可以通过以下脚本实现前两个步骤。

```
# Chap06/stack_classification_save_model.py
import json
import pickle
from datetime import datetime
from argparse import ArgumentParser
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import MultiLabelBinarizer

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--questions')
  parser.add_argument('--output')
  parser.add_argument('--max-df', default=1.0, type=float)
  parser.add_argument('--min-df', default=1, type=int)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  stop_list = stopwords.words('english')

  all_questions = []
  all_labels = []
  with open(args.questions, 'r') as f:
    for line in f:
```

```
        doc = json.loads(line)
        question = "{} {}".format(doc['Title'], doc['Body'])
        all_questions.append(question)
        all_labels.append(doc['Tags'].split(' '))

    vectorizer = TfidfVectorizer(min_df=args.min_df,
                                 stop_words=stop_list,
                                 max_df=args.max_df)
    X = vectorizer.fit_transform(all_questions)
    mlb = MultiLabelBinarizer()
    y = mlb.fit_transform(all_labels)

    classifier = OneVsRestClassifier(LinearSVC())

    classifier.fit(X, y)
    model_to_save = {
      'classifier': classifier,
      'vectorizer': vectorizer,
      'mlb': mlb,
      'created_at': datetime.today().isoformat()
    }
    with open(args.output, 'wb') as f:
      pickle.dump(model_to_save, f)
```

以上代码与前一节中执行的线下分类非常相似。主要差别是，训练分类器后，我们并不用它预测任何事情，而是将它存在一个文件中。

输出文件的打开方式是 `wb` 模式，也就是说，文件指针有对文件的写权限，并且这个文件是一个二进制文件，而不是纯文本文件。pickle 包将复杂的 Python 对象序列化为字节码，然后存放在一个文件中。

该模块的界面和 `json` 包非常相似，通过简单的 `load()` 和 `dump()` 函数执行 Python 对象和文件间的转换。我们不仅仅需要分类器，还需要向量器和 `MultiLabelBinarizer` 实例。因此，我们使用 `model_to_save` 变量，该变量是一个 Python 字典，用于存放所有需要的数据。该字典的 `created_at` 键包含当前系统的 ISO 8601 格式的时间，当需要追踪分类器的不同版本或者有新的数据并决定重新训练分类器时，该时间很有用处。

可以用以下命令运行上述脚本。

```
$ python stack_classification_save_model.py \
  --questions movies.questions4classification.jsonl \
  --min-df 5 \
  --output questions-svm-classifier.pickle
```

运行后会生成 questions-svm-classifier.pickle 文件，其他应用可以使用该文件。

为了简洁起见，我们将从命令行实现一个基本的界面。同样的方法可以与 Web 应用（例如，使用前面提到的 Flask）非常简单地整合，以通过 Web 界面向用户提供预测功能。

stack_classification_user_input.py 脚本实现了用户交互应用。

```
# Chap06/stack_classification_user_input.py
import sys
import json
import pickle
from argparse import ArgumentParser

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--model')
  return parser

def exit():
  print("Goodbye.")
  sys.exit()

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  with open(args.model, 'rb') as f:
    model = pickle.load(f)
  classifier = model['classifier']
  vectorizer = model['vectorizer']
  mlb = model['mlb']

  while True:
    print("Type your question, or type \"exit\" to quit.")
    user_input = input('> ')
    if user_input == 'exit':
      exit()
    else:
      X = vectorizer.transform([user_input])
      print("Question: {}".format(user_input))
      prediction = classifier.predict(X)
      labels = mlb.inverse_transform(prediction)[0])
      labels = ', '.join(labels)
      if labels:
        print("Predicted labels: {}".format(labels))
      else:
        print("No label available for this question")
```

该脚本用 ArgumentParser 读入命令行参数--model，用于传递前面序列化过的 pickle 文件。

脚本还用前面介绍过的 pickle 接口导入 pickle 文件。该文件是一个二进制文件，因此可以用 rb 模式（r 表示 read，b 表示 binary）打开。然后我们就可以接入前面生成的分类器、向量器和多标签二进制表示器。

接着该应用进入无限的 while True 循环，持续询问用户的输入。如果运行以下命令：

```
$ python stack_classification_user_input.py \
  --model questions-svm-classifier.pickle
```

6

我们将看到以下提示：

```
Type your question, or type "exit" to quit.
>
```

写入一些输入后，input()函数会将这些输入读入 user_input 变量。该变量用于检查用户是否输入了会导致应用退出的单词 exit。如果没有该单词，分类器将对用户输入的任何文本执行分类。

预测步骤需要将输入转变为特征向量。这是由向量化器执行的。值得重视的是，vectorizer.transform()函数将一个列表（或一个迭代器）作为参数，而不是将 user_input 字符串直接作为输入。可以将向量化后的输入传入分类器的 predict()函数，然后执行分类。

接着将分类器的结果以可读的格式返回给用户。这一阶段的预测实际上是一个 NumPy 数组，表示有效标签列表的一个二进制掩码。MultiLabelBinarizer 负责在训练阶段生成这个掩码，然后用 inverse_transform()函数将二进制掩码映射回原始标签。

这个调用的结果是一个元组列表。准确来说，因为只传递了一个文件来进行分类，所以该列表只有单个元组（因此在 inverse_transform()函数中使用[0]索引）。

向用户展示结果前，如果标签的元组不为空（可能出现一个自定义消息），就将其连接为一个字符串。

现在可以查看分类器的实际运行情况。

```
$ python stack_classification_user_input.py \
  --model questions-svm-classifier.pickle
Type your question, or type "exit" to quit.
> What's up with Gandalf and Frodo lately? They haven't been in the Shire
for a while...
Question: What's up with Gandalf and Frodo lately? They haven't been in the
Shire for a while...
Predicted labels: plot-explanation, the-lord-of-the-rings
Type your question, or type "exit" to quit.
> What's the title of the latest Star Wars movie?
Question: What's the title of the latest Star Wars movie?
Predicted labels: identify-this-movie, star-wars, title
Type your question, or type "exit" to quit.
> How old is Tom Cruise?
Question: How old is Tom Cruise?
No label available for this question
Type your question, or type "exit" to quit.
> How old is Brad Pitt?
Question: How old is Brad Pitt?
Predicted labels: character, plot-explanation
Type your question, or type "exit" to quit.
> exit
Goodbye.
```

调用这个脚本后，提示让我们输入自己的问题。高亮的代码显示了用户输入和系统回应。

非常有趣的是，该分类器可以选取《魔戒》中的一些名字（Frodo、Gandalf 和 Shire），并在没有提及标题的情况下准确为问题打上标签。出于某些原因，该问题也被理解为对画图解释的请求。更准确地说，问题的某些特征导致分类器将该文档标记为属于画图解释类。

第二个输入包含《星球大战》短语，因此识别该标签非常简单直接。该问题也问了标题，因此有两个额外的类名称 `title` 和 `identify-this-movie`，它们都与该问题相关。从目前来看，分类器还比较准确。

对于最后两个问题，我们可以看到，分类器并不总是很完美。当问到 Tom Cruise，分类器并没有对这个问题给定任何标签。这可能是由于训练数据中缺少有趣的特征，即 tom 和 cruise 这两个单词。Brad Pitt 的结果稍好一点，这个问题与 `character`（一点点扩展）和 `plot-explanation`（明显错误）两个标签相关。

最后用户输入 `exit`，应用退出。

以上实验表明，分类器的准确率还远不够完美。在实际应用中，我们测试了分类器的能力。虽然可以通过模糊输入迷惑分类器或输入分类器未训练过的单词，但本节的核心思想是我们并不局限于学术实验来测试模型。将机器学习模型整合到合适的用户应用是一个非常直接的任务，特别是用 Python 优雅简单的界面。

实时的分类应用数不胜数：

- 情感分析可以自动识别用户在评论中的观点并在线显示
- 垃圾邮件过滤能够拦截或者过滤潜在的垃圾信息
- 不敬词检测可以识别在论坛上发布不雅消息的用户，并自动拦截该消息

如果感兴趣，你可以重用第 5 章中介绍的 Flask 并扩展这些内容，以创建一个简单的 Web 应用来导入一个机器学习模型，并用它实时对用户问题进行分类。

6.5　小结

本章介绍了 Web 上非常流行的问答应用。Stack Exchange 以及 Stack Overflow 在程序员圈的流行是由社区中的高质量内容驱动的。本章介绍了如何与 Stack Exchange API 交互，以及如何用 Stack Exchange 数据存档获得 Stack Exchange 的全部数据集。

本章的第二部分介绍了分类任务和解决问题的相关监督机器学习方法。Stack Exchange 上的标签数据提供了创建预测模型的可能性。本章的用例是预测问题的标签，该技术可以用于很多应用。本章的最后一部分进一步扩展，展示了如何将机器学习模型轻松整合到实时的用户应用中。

下一章将关注博客，介绍自然语言处理技术的应用。

第 7 章

博客、RSS、维基百科和自然语言处理

本章重点关注**自然语言处理**（NLP），它是研究如何处理自然语言的领域。在深入了解自然语言处理的细节之前，我们先介绍一些从 Web 上下载文本数据的方法。

本章将介绍以下主题：

- 如何与 WordPress.com 和 Blogger API 交互
- 网站订阅格式（RSS 和 Atom）及其使用方法
- 如何以 JSON 格式存储来自博客的数据
- 如何与维基百科 API 交互以搜索关于实体的信息
- 自然语言处理的核心概念，特别是如何进行文本预处理
- 如何处理文本数据以识别文本中提到的实体

7.1 博客和自然语言处理

如今，博客是 Web 的重要组成部分，而且是广受欢迎的社会媒体平台。企业、专业人士和有各类爱好的人用博客接近受众，以促销产品和服务或讨论有趣的话题。得益于丰富的网络发布工具和服务，非技术用户可以轻松地在几分钟内发布内容，建立个人网站。

从数据挖掘者的视角来看，博客是实现文本挖掘的完美平台。本章将重点介绍两个话题：如何从博客中获取文本数据，以及如何对这些数据应用自然语言处理技术。

虽然自然语言处理是一个复杂的领域，需要不止一本书的内容来详细介绍，但我们将从实用主义的角度介绍一些基本理论和实际案例。

7.2 从博客和网站获取数据

目前，丰富的网站中有很多有趣的文章，查找待挖掘的文本数据应该不是一个大问题。每次

手动保存一篇文章显然不能很好地扩大数据的规模，因此，本节将介绍一些方法对从网站获取数据的过程进行自动化。

首先，我们将介绍两个流行的博客服务——WordPress.com 和 Blogger，它们都提供了 API 与其平台交互。其次，我们将介绍 RSS 和 Atom 网站标准，这两个标准被很多博客和新闻发布者用来传播易于被计算机读取的内容。最后，我们将简要介绍更可行的选项。例如，与维基百科连接或者在没有其他选项时使用网络爬虫。

7.2.1 使用 WordPress.com API

WordPress.com 是由开源的 WordPress 软件驱动的博客和网站主机提供商。该服务为注册用户提供免费的博客、付费升级和其他付费服务。如果用户只需要阅读和评论博文内容，那么无须注册，除非博主特别指定（博主可以将博文标记为仅自己可见）。



WordPress.com 和 WordPress.org

WordPress 的新用户经常混淆这两者。开源的 WordPress 软件由 WordPress.org 开发和分发，你可以下载该软件的副本并安装在自己的服务器上。另一方面，WordPress.com 是一个主机提供商，为不想安装软件的用户提供定制化的解决方案，以处理各种服务器配置。WordPress.com 上博客的域名通常是 blog-name.wordpress.com 形式，除非博主使用付费的域名。

本节讨论的 API 由 WordPress.com 提供，因此我们可以获取的数据是由该提供商的主机获取的。

Python 还没有对 WordPress.com API 较好的客户端支持。这为我们提供了直接与 API 交互的机会，即可以编写自己的客户端。

好在完成这个任务所需的努力并不过重，Python 的 requests 库为我们处理 HTTP 提供了非常简便的接口。我们将用它建立对 WordPress.com API 的一系列调用，调用的目的是从特定域名下载一些博文。

第一步是在我们的虚拟环境中安装这个库。

```
$ pip install requests
```

第二步是查看文档，特别是`/sites/$site/posts` 端点（https://developer.wordpress.com/docs/api/1.1/get/sites/%24site/posts/）。

以下脚本定义了`get_posts()`函数，该函数用于查询 WordPress.com API 并处理翻页。

```
# Chap07/blogs_wp_get_posts.py
import json
from argparse import ArgumentParser
```

```
import requests

API_BASE_URL = 'https://public-api.wordpress.com/rest/v1.1'

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--domain')
  parser.add_argument('--posts', type=int, default=20)
  parser.add_argument('--output')
  return parser

def get_posts(domain, n_posts=20):
  url = "{}/sites/{}/posts".format(API_BASE_URL, domain)
  next_page = None
  posts = []
  while len(posts) <= n_posts:
    payload = {'page_handle': next_page}
    response = requests.get(url, params=payload)
    response_data = response.json()
    for post in response_data['posts']:
      posts.append(post)
    next_page = response_data['meta'].get('next_page', None)
    if not next_page:
      break
  return posts[:n_posts]

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  posts = get_posts(args.domain, args.posts)

  with open(args.output, 'w') as f:
    for i, post in enumerate(posts):
      f.write(json.dumps(post)+"\n")
```

以上脚本用 `ArgumentParser` 接收命令行参数。它还有两个必备参数：`--domain` 和 `--output`，前者用于设置我们检索数据的域名，后者用于设置 JSON Lines 文件的名称。可选参数是`--posts`，表示我们从给定域名检索的博文数量，其默认值为 20。

域名是指定博客的完整 URL，如 your-blog-name.wordpress.com。

例如，可以用以下命令运行脚本。

```
$ python blogs_wp_get_posts.py \
  --domain marcobonzanini.com \
  --output posts.marcobonzanini.com.jsonl \
  --posts 100
```

几秒后，脚本会生成 JSON Lines 文件，其中每行都是以 JSON 格式表示的一篇博文。

在描述输出格式前，我们来详细分析一下 `get_posts()`函数，该函数是我们脚本的核心。

该函数有两个参数：domain 和 n_posts，前者是博客的域名，后者是我们希望检索的博文数量，默认值为 20。

首先，该函数为相关 API 端点定义了 URL，并初始化了 next_page 变量，用于迭代多个页面的结果。默认情况下，API 在每个页面返回 20 个结果（即 20 篇博文）。如果博文的数量大于这个值，就需要迭代多个页面。

该函数的核心是 while 循环，用于每次检索一个页面的结果。我们用 GET 方法调用 API 端点，该端点接收文档中为其定义的几个参数。page_handle 参数用于指定我们感兴趣的页面结果。第一次迭代时，该变量的值是 None，因此检索从开头开始。用 response.json()方法返回的 JSON 格式的响应数据包含博文列表 response_data['posts']，以及一些存储在 response_data['meta']中关于请求的元数据。如果有下个页面的结果，那么元数据的字典将包含一个 next_page 键。如果该键没有出现，则将其设置为 None。

当没有 next_page（即已经下载了所有可用的博文）或者已经达到足够的博文数量（以 n_posts 指定）时，循环就会停止。最后，我们将返回博文的列表切片。当需要的博文数量 n_posts 不是页面大小的整数倍时，切片是很有必要的。例如，如果我们指定需要的博文数量为 30，那么这段代码将下载两页博文，其中每个页面有 20 篇博文，因此切片需要切除最后 10 项。

每个博文对象由表 7-1 中的属性来定义。

表 7-1　博文对象的属性

属　性	描　述
ID	博客中博文的 ID
URL	博文的完整 URL
attachment_count	附件的数量（如媒体文件等）
attachments	附件对象列表
author	作者的个人资料对象，其中包括全名、登录名、头像、头像资料 URL 等
categories	博文所属的分类列表
content	博文的完整内容，包括 HTML 标记
date	ISO 8601 格式的博文发布时间
discussion	有关评论、ping 等的信息
excerpt	博文的简短摘要，通常是前几句话
feature_image	特征图像的 URL，如果特征图像出现的话
global_ID	博文的全局 ID
guid	博客域名和博文 ID 合并为一个合法的 URL
i_like	表示登录用户是否喜欢该博文
is_following	表示登录用户是否关注了该博客
is_reblogged	表示登录用户是否转载了该博文
like_count	喜欢该博文的用户数量

（续）

属　性	描　述
meta	来自 API 的元数据，如用于自动 API 发现的超媒体链接等
modified	ISO 8601 格式的博文最后修改时间
short_url	用于社会媒体和移动端共享的短 URL（http://wp.me）
site_ID	博客的 ID
slug	博文的 URL 识别
status	博文的当前状态，如已发布、草稿、进行中，等等
sticky	表示博文是否固定（不考虑发布时间始终置顶）
tags	与博文相关的标签列表
title	博文的标题

正如我们看到的，博文的结构比其内容复杂得多，但对本章来说，我们主要关注文本内容，包括标题和摘要。

7.2.2　使用 Blogger API

Blogger 是另一个流行的博客发布服务，于 2003 年被 Google 收购。Blogger 中的博客通常在 blogspot.com 的子域名下，如 your-blog-name.blogspot.com。有几个国家使用了特定国家的域名（如英国的 blogspot.co.uk），也就是说，特定国家的用户会以透明的方式被重定向到指定域名。

Blogger 是 Google 的产品，其账户是集中式的。也就是说，只要注册过 Google 账户，用户就可以自动接入 Blogger。这同样适用于 Blogger API。作为开发者，我们可以通过 Google Developers Console 接入 Blogger，这与第 5 章中的做法相同，我们创建了一个可以接入任何 Google API 的项目。可以重用同样的项目并在 Developers Console 中启用 Blogger API，也可以重新创建一个项目。具体步骤参见第 5 章中的介绍。在这两种方式中，最重要的步骤就是**为项目启用 Blogger API**，否则无法接入权限。

在处理 Google Plus API 时，我们还讨论了接入键的创建，特别是服务器键，这也是我们将用来接入 Blogger API 的键。如果依照第 5 章中的步骤来做，我们应该已经有了一个可用的服务器键，可以将其存储在环境变量中。

```
$ export GOOGLE_API_KEY="your-api-key-here"
```

对于客户端库，可以使用与第 5 章中相同的客户端库，因为它是所有 Google API 的通用接口。如果还未安装它，可以使用 `pip` 将其添加到我们的虚拟环境中。

```
$ pip install google-api-python-client
```

配置好环境后，就可以开始编写与 Blogger API 交互的脚本，以便从给定的博客中下载一些博文。

```
# Chap07/blogs_blogger_get_posts.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--url')
  parser.add_argument('--posts', type=int, default=20)
  parser.add_argument('--output')
  return parser

class BloggerClient(object):

  def __init__(self, api_key):
    self.service = build('blogger',
                         'v3',
                         developerKey=api_key)

  def get_posts(self, blog_url, n_posts):
    blog_service = self.service.blogs()
    blog = blog_service.getByUrl(url=blog_url).execute()
    posts = self.service.posts()
    request = posts.list(blogId=blog['id'])
    all_posts = []
    while request and len(all_posts) <= n_posts:
      posts_doc = request.execute()
      try:
        for post in posts_doc['items']:
          all_posts.append(post)
      except KeyError:
        break
      request = posts.list_next(request, posts_doc)
    return all_posts[:n_posts]

if __name__ == '__main__':
  api_key = os.environ.get('GOOGLE_API_KEY')
  parser = get_parser()
  args = parser.parse_args()

  blogger = BloggerClient(api_key)

  posts = blogger.get_posts(args.url, args.posts)

  with open(args.output, 'w') as f:
    for post in posts:
      f.write(json.dumps(post)+"\n")
```

编排与所有 Google API 交互的 Google API 客户端有一个服务构建器的概念，这是一种用于创建服务对象的工厂方法，用于查询 API。为了提供易于使用的接口，我们将创建一个 `BloggerClient` 类来保留 API 键、创建服务对象并提供一个 `get_posts()` 函数，这与我们为 WordPress.com API 所定义的类似。

可以从命令行传入三个参数来调用使用了 ArgumentParser 的脚本，如下所示。

```
$ python blogs_blogger_get_posts.py \
  --url http://googleresearch.blogspot.co.uk \
  --posts 50 \
  --output posts.googleresearch.jsonl
```

与 WordPress.com 类似，我们用--posts 和--output 参数来分别定义期望的博文数量和输出文件的名称。与前面的示例不同，--url 参数要求给定的字符串是博客的完整 URL，而且不能只是域名（即应该包含 http://）。通过使用前面的命令，我们将获取 Google Research 博客中最新的 50 篇博文。

进一步查看 BloggerClient 类，该构造器只接收一个参数，即开发者在 Google Developers Console 注册应用时获得的 API 键。它使用 build()工厂函数创建 service 对象。需要注意的是，我们使用的是第 3 版的 API——v3，因此需要查看对应版本的文档。

get_posts()方法完成繁重的工作。该函数的布局与 WordPress.com API 中定义的 get_posts()函数非常相似，都带有博客 URL 和需要的博文数量这两个参数。

与 WordPress.com API 不同的是，Blogger API 首先需要一个博客对象，因此我们需要将 URL 转换成数值 ID。这是通过该对象的 getByUrl()函数执行的，该函数是与 self.service.blogs()同时创建的。

另一个有趣的不同之处是，与懒惰评估函数类似，对大多数服务函数的调用并不会直接触发对 API 的调用，因此我们需要特别指定一个 execute()函数来执行 API 请求并获得响应。

request.execute()调用创建了一个 posts_doc 对象，该对象可能包含、也可能不包含 item 键（博文列表）。将其包装到 try/except 模块中可以确保，如果特定的调用未返回任何博文，该迭代就会中断停止，但不会抛出任何异常。

一旦达到需要的博文数量或者没有其他页面可用，就用切片技术返回博文列表。

Blogger API 返回的 JSON 文档比 WordPress.com 的更简单，但所有的键属性都会出现。表 7-2 总结了每篇博文的主要属性。

表 7-2　博文的属性

属　性	描　述
author	表示博文作者的对象，其中包括显示名称、用户 ID、资料图像和 URL
blog	表示博客本身的对象（如博客 ID）
content	博文的完整内容，包括 HTML 格式
id	博文的 ID
labels	与博文相关的标签列表

（续）

属　性	描　述
published	ISO 8601 格式的博文发布日期
replies	表示博文评论/回复的对象
selfLink	博文的 API 链接
title	博文的标题
updated	ISO 8601 格式的博文最后更新日期
url	博文的 URL

除博文的内容外，以上还可以看到更多有趣的信息，但本章重点关注文本数据。

7.2.3 解析 RSS 和 Atom 订阅

很多博客（也可以说是很多网站）都以标准格式向外提供其内容，但这种格式并不是提供给终端用户在屏幕上阅读的，而是便于第三方出版商解析。这就是需要 RSS（rich site summary，简易信息聚合）和 Atom 的原因，我们可以用这两种基于 XML 的格式从实现该特性的网站上快速获取各种信息。

RSS 和 Atom 属于网站订阅格式家族，常用于为用户提供网站内容的更新。订阅提供商汇聚订阅源，意味着用户可以通过一个应用订阅特定的订阅源，一旦有新的内容发布，就可以获得更新。包括邮件阅读器在内的一些应用都提供了将网站订阅整合到工作流的特征。

在挖掘博客和文章的场景中，订阅非常有趣，因为订阅为给定网站提供了一个进入项，即以机器可读的格式提供网站内容。

例如，**英国广播公司**（BBC）提供了广泛的新闻订阅源，分为多个主题：世界新闻、技术、体育等。例如，置顶新闻的订阅源为（浏览器可读）http://feeds.bbci.co.uk/news/rss.xml。

Python 对读取网站订阅源的支持非常直接。我们需要做的是安装订阅源解析库，该库会下载订阅源并将 XML 解析为 Python 对象。

首先，在虚拟环境中安装这个库。

```
$ pip install feedparser
```

以下脚本用于下载给定 URL 的订阅源，并用常规的 JSON Lines 格式保存新闻项。

```
# Chap07/blogs_rss_get_posts.py
import json
from argparse import ArgumentParser
import feedparser

def get_parser():
```

```
  parser = ArgumentParser()
  parser.add_argument('--rss-url')
  parser.add_argument('--json')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  feed = feedparser.parse(args.rss_url)
  if feed.entries:
    with open(args.json, 'w') as f:
      for item in feed.entries:
        f.write(json.dumps(item)+"\n")
```

该脚本接收通过 `ArgumentParser` 定义的两个参数：`--rss-url` 和`--json`，前者用于传递订阅源的 URL，后者用于指定以 JSON Lines 格式保存的订阅源的文件名称。

例如，为了从 BBC 置顶新闻下载 RSS 订阅源，可以使用以下命令。

```
$ python blogs_rss_get_posts.py \
  --rss-url http://feeds.bbci.co.uk/news/rss.xml \
  --json rss.bbc.jsonl
```

这个脚本非常易懂，最复杂的工作由 feedparser 库的 `parse()`函数完成。该函数识别该格式并将 XML 解析为一个对象，该对象包含 `entries` 属性（新闻项的一个列表）。通过迭代该列表，可以用几行代码将每个项存入我们希望的 JSON Lines 格式。

这些新闻项的有趣属性如下所示。

- `id`：新闻项的 URL
- `published`：发布时间
- `title`：新闻标题
- `summary`：新闻的简短摘要

7.2.4　从维基百科获取数据

维基百科可能不需要过多介绍，它是最流行的网站和参考工具之一。2015 年年底，维基百科的英文文章数量达到了约 500 万篇，所有语言的文章共有 3800 多万篇。

维基百科以 API 的方式提供内容。它也定期提供完整的数据存档。

一些项目提供了维基百科 API 的 Python 包装器。一个特别实用的库叫作 wikipedia。可以用常规方式将其安装到虚拟环境中。

```
$ pip install wikipedia
```

这个库的接口非常直接，正如其文档所说，我们从而可以关注如何使用来自维基百科的数据，而不是如何获取这些数据。

除了获取完整的页面，维基百科 API 还提供了搜索和自动摘要等特性。我们来查看以下示例。

```
>>> import wikipedia
# 获得维基百科页面
>>> packt = wikipedia.page('Packt')
>>> packt.title
'Packt'
>>> packt.url
'https://en.wikipedia.org/wiki/Packt'
# 获得页面的摘要
>>> wikipedia.summary('Packt')
"Packt, pronounced Packed, is a print on demand publishing company based in
Birmingham and Mumbai." # 更长的描述
```

在以上示例中，我们直接获得了 Packt 出版社的页面，并且知道 `Packt` 这个名称是唯一的。

当不确定我们感兴趣的实体的精确页面时，维基百科 API 的搜索和模糊特征就非常有用了。

例如，`London` 至少与两个城市（一个在英国，另一个在加拿大）和几个实体或事件（如伦敦塔、伦敦大火等）有关。

```
>>> wikipedia.search('London')
['London', 'List of bus routes in London', 'Tower of London', 'London,
Ontario', 'SE postcode area', 'List of public art in London', 'Bateaux
London', 'London Assembly', 'Great Fire of London', 'N postcode area']
```

如果存在拼写错误，那么结果会出人意料。

```
>>> wikipedia.search('Londn')
['Ralph Venning', 'Gladys Reynell', 'GestiFute', 'Delphi in the Ottoman
period']
```

可以通过询问维基百科的建议来解决上述问题。

```
>>> wikipedia.suggest('Londn')
'london'
```


这里以伦敦为例，我们发现该实体的摘要特别长。假如需要更短的摘要，可以通过 API 指定句子的数量，如图 7-1 所示。

```
>>> wikipedia.summary('London', sentences=2)
'London /ˈlʌndən/ is the capital and most populous city of England
and the United Kingdom. Standing on the River Thames in the south
east of Great Britain, London has been a major settlement for two
millennia.'
```

图 7-1　简短摘要

最后，我们思考一下消除歧义的问题。当访问指定名称的维基百科页面时，如果匹配到多个实体，而没有一个确定的候选项时，那么就会出现一个消除歧义匹配问题的页面。

从 API 的角度来看，该结果是一个异常，如下所示。

```
>>> wikipedia.summary('Mercury')
Traceback (most recent call last):
    # 很长的错误消息
wikipedia.exceptions.DisambiguationError: "Mercury" may refer to:
Mercury (element)
Mercury (planet)
Mercury (mythology)
# ……很长的"Mercury"建议列表
```

为了避免该问题的出现，可以将请求包装在一个 `try/except` 模块中。

```
>>> try:
...   wikipedia.summary('Mercury')
... except wikipedia.exceptions.DisambiguationError as e:
...   print("Many options available: {}".format(e.options))
...
Many options available: ['Mercury (element)', 'Mercury (planet)', 'Mercury
(mythology)', ...] # 长列表
```

接入维基百科可以下载完整文章，但这并不是其唯一的用处。我们将介绍如何从文本中抽取名称和命名实体。此外，还可以用维基百科来增强命名实体的抽取结果，增强的方式就是用简短的摘要解释识别出的实体。

7.2.5　关于网络爬取的一点建议

网络爬取是自动从网站抽取信息并下载的过程。网络爬取软件基本上是模拟人在网站上浏览信息（并保存），目的是获取大量数据或执行需要自动交互的特定任务。

当 Web 服务不提供 API 或订阅源下载时，构建网络爬虫有时就是解决方案。

虽然网络爬虫是正当合理的应用，但需要注意的是，自动爬取可能会违背目标网站所规定的使用条款。更重要的是，还需要考虑一些道德和法律的问题（可能随国家而不同），举例如下。

- 是否有权限访问并下载数据？
- 是否过量下载了网站数据？

通常可以查看网站的使用条款来回答第一个问题。第二个问题比较难以回答。从原则上来说，如果与网站进行大量交互（例如，在短时间内获取大量页面），我们肯定超过了普通用户的合理交互，可能会导致网站的服务性能下降。如果我们连接的网站处理不了这样大量的请求，甚至会引发拒绝服务攻击。

在有大量可用数据以及很多服务让这些数据很容易获取的情况下，构建网络爬虫应该是 API 不可用时的最后选择。

这个话题比较复杂。虽然有为爬虫量身定做的工具 Scrapy，但还是可以用 requests 和 Beautiful Soup 等库实现定制化的解决方案。这个话题超出了本书的范围，如果感兴趣，你可以找到很多相关出版物，如 Richard Lawson 撰写的《用 Python 写网络爬虫》。

7.3 自然语言处理基础

本节介绍自然语言处理这个复杂领域的基础。前面的章节介绍了处理文本数据的一些基础（如分词）。这里我们将尝试进一步理解该技术。由于复杂性以及其他方面的因素，我们将从实用主义入手，只介绍一些理论基础以辅助实际示例。

7.3.1 文本处理

任何自然语言处理系统的必备部分都是预处理流水线。对一段文本执行有趣的任务前，我们需要先将其转换为有用的表示。

前面在没有深入了解文本预处理的情况下做了一些文本数据分析，并以常用工具的实用方法实现。本节将特别介绍一些常见的预处理步骤，并讨论它们在构建自然语言处理系统中的作用。

1. 句子边界检测

给定一段文本（即一个字符串），将其分割为句子列表的任务称作**句子边界检测**（也称作**句子末尾检测**或**句子分词**）。

句子是将单词有意义地组织起来并在语法上表达完整想法的语言单元。从语言学（语言的科学研究）的角度来看，彻底讨论定义句子的所有相关方面可能需要较长篇幅。从实用性和构建自然语言处理系统的角度来看，上面语言单元的通用概念已经足够了。

需要注意的是，虽然句子是有意义的独立单元，但有时也需要作为大段内容（如一个段落或章节）的一部分。这是因为一些句子中的单词可能会指向其他句子中表达的概念。思考以下示例：

> Elizabeth II is the Queen of the United Kingdom. She was born in London in 1926 and her reign began in 1952.

以上这段简单的文本包含两个句子。第二个句子使用了代词 she 和 her。如果单独来看这个句子，这两个单词并不能表明句子的主语是谁。有了上下文，读者可以很轻松地将这两个单词和前面句子的主语 Elizabeth II 联系起来。这是一个非常难的自然语言处理问题，名叫**指代消解**

（anaphora resolution）。通常来说，与其他句子链接的句子经常利用局部上下文，也就是说，链接的是上一个/下一个句子，而不是文本中相隔较远的句子。识别更广上下文（如段落或章节）的边界在有些应用中非常有用，但句子边界识别是一个常用的起点。

从实际应用来看，分割句子并不是一个很大的问题，因为我们可以使用标点符号（如句号、感叹号等）来识别句子的边界。这种过于简化的方法会导致一些错误，如下例所示。

```
>>> text = "Mr. Cameron is the Prime Minister of the UK. He serves since
2010."
>>> sentences = text.split('.')
>>> sentences
['Mr', ' Cameron is the Prime Minister of the UK', ' He serves since 2010',
'']
```

正如我们所看到的，将每个**点**作为句号会将 Mr. Cameron 分割为两个部分，这样得到的是一个没有意义的句子 Mr（以及最后一个空的句子）。还可以列举很多其他示例，主要是一些缩略语（如 Dr.、Ph.D、U.S.A.等）。

好在 NLTK 提供了一个简单的解决方案，如下所示。

```
>>> from nltk.tokenize import sent_tokenize
>>> sentences = sent_tokenize(text)
>>> sentences
['Mr. Cameron is the Prime Minister of the UK.', 'He serves since 2010.']
```

这里的句子都被准确识别了。该示例非常简单，但在有些情况下，NLTK 并不会执行正确的句子识别。通常来说，该库可以较好地分割句子，因此多数应用场景下无须过多担心。

2. 单词分词

将给定句子（即一个字符串）分割成单词列表的任务称作**单词分词**，有时简称为**分词**。此时，单词通常称作**词项**。

对包括英语在内的大多数欧洲语言来说，分词看起来是一个非常直观的任务，因为单个单词由空格分隔。这种过于简化的方式也有缺点。

```
>>> s = "This sentence is short, nice and to the point."
>>> wrong_tokens = s.split()
>>> wrong_tokens
['This', 'sentence', 'is', 'short,', 'nice', 'and', 'to', 'the', 'point.']
```

这里的 `short,`和 `point.`（注意结尾的标点符号）明显不正确：分词失败的原因是，简单地用空格分隔并没有考虑到标点符号。构建分词器来考虑一种语言的所有细节是难以实现的。此外，对于德语或芬兰语等一些欧洲语言来说，这实现起来更加困难，因为这些语言中包含大量的组合词，如两个及更多单词连接起来组成一个更长的单词，且含义可能与其组成部分类似、也可能大不相同。更具挑战的是一些亚洲语言，如中文在书写时是没有空格符号作为分隔的。一些特

殊的库可以处理特定的语言。例如，jieba 是一个 Python 库，专门用于中文的单词分割。

你并不需要担心语言处理的这些方面，因为我们现在只关注英语的处理。此外，NLTK 包已经具备了标准英语的多种处理方法。

```
>>> from nltk.tokenize import word_tokenize
>>> s = "This sentence is short, nice and to the point."
>>> correct_tokens = word_tokenize(s)
>>> correct_tokens
['This', 'sentence', 'is', 'short', ',', 'nice', 'and', 'to', 'the',
'point', '.']
```

可以看到，标点符号也作为词项输出，可用于后面的处理。

正如我们在本示例中看到的，单个单词都被准确识别了。标点符号也作为词项输出，可用于后面的处理。NLTK 的 `word_tokenize()` 函数需要安装 punkt 包。如果该函数抛出异常，可以参考第 1 章，其中提供了可以解决该问题的配置方法。

3. 词性标注

一旦有了单词序列，就可以标注每个单词的语法类别。该过程叫作**词性标注**（part-of-speech tagging），词性标注在很多文本分析任务中都有广泛的应用。

常见的语法类别包括名词、动词和副词，每一种都是一个不同的标签符号。不同的标签器基于不同的标签集合，这也就意味着不同的标签器有不同的可选标签，且每个标签器的输出各不相同。默认情况下，NLTK 库使用的是来自 Penn Treebank Project 的标签集合。

我们可以从在线帮助获取完整的可用标签列表及其含义。

```
>>> nltk.help.upenn_tagset()
```

这将产生一个长输出，其中包括所有的标签及其描述，以及一些示例。最常用的标签有 `NN`（常见名词）、`NNP`（专有名词）、`JJ`（形容词）和 `V*`（一些以 V 开头的标签，表示动词的不同时态）。

为了将单词与相关的标签关联起来，可以使用 `nltk.pos_tag()` 函数。NLTK 提供了多个词性标注方法的实现。在最新版本的工具中，默认的词性标注器是平均感知器（averaged perceptron）。正如第 1 章中所介绍的，NLTK 中的一些模型需要通过 `nltk.download()` 界面下载一些额外的数据。平均感知器模型同样需要下载，如图 7-2 所示。

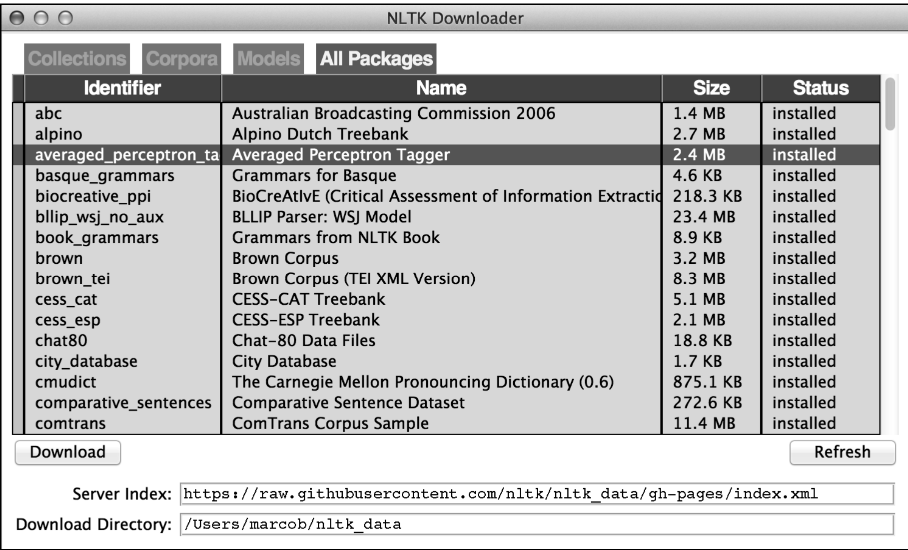


图 7-2　NLTK 下载界面高亮了用于词性标注的平均感知器模型

确保安装好相关的模型后，可以在一些示例句子上测试 `pos_tag()` 函数。

```
>>> from nltk import pos_tag
>>> tokens = word_tokenize("This sentence is short, nice and to the point")
>>> pos_tag(tokens)
[('This', 'DT'), ('sentence', 'NN'), ('is', 'VBZ'), ('short', 'JJ'), (',',
','), ('nice', 'JJ'), ('and', 'CC'), ('to', 'TO'), ('the', 'DT'), ('point',
'NN')]
```

`pos_tag()` 函数的输出是一个元组列表，其中每个元组都是 `(token, tag)` 格式。例如，第一个词项 `This` 是一个限定词，第二个词项 `sentence` 是一个常见名词等。

词性标注的一个常见问题是，虽然研究人员对一些基本的类别达成了共识，但并没有一个完全正确的标签集合，因此使用不同的标签器会产生不同结果。由于自然语言的模糊性和细微差别，一些单词在不同语境下的标签是不同的。实际上，很多英语单词可以分到多个类别，例如单词 fish、walk 和 view 既是动词又是名词。

词性标注通常作为最后一步处理。具体来说，探索词性信息至少在两种情况下非常有用：查找单词的原始词干（查看下面的**单词归一化**）时，以及尝试从非结构化文本中抽取结构化信息（参见 7.3.2 节）时。

4. 单词归一化

分词和词性标注在很多应用中都是常见的预处理步骤。根据不同的任务，可能需要应用一些额外的步骤来提高应用的准确度。

这里我们将讨论**单词归一化**，以便对单个词执行一些额外的步骤。

- **大小写归一化**

为了突出文本归一化的需求，我们从以下的简单示例开始。

```
>>> "president" == "President"
False
```

虽然对于大多数程序员来说，`president` 和 `President` 显然是两个不同的字符串，但是从语言学的角度来看，通常需要将这两个单词作为相同单词。例如，当计算频率时，我们不希望对同样的两个单词分别计算频率。

实现该需求的方法之一是执行大小写转换，将整个文本转换为小写或大写。

```
>>> "president".lower() == "President".lower()
True
```

在这个示例中，我们通过小写转换对所有单词进行转换，以便同样的词可以被当成一个。

需要注意的是，转换不可逆。如果用小写版本覆盖了文本，那么不能重现原始文本。在很多应用中，对文本进行归一化时，可能需要向用户显示原始文本。这种情况下，保留原始文本并将其作为不可变的数据非常重要。

另外还需要注意的是，文本归一化并不总能带来期望的结果。典型的示例是 United States 的首字母缩写形式 US，该缩写归一化后会映射为人称代词 us。进行试错对于发现其他错误案例是必要的。

- **词干提取**

在英语和其他一些语言中，有些单词在形式上不同，但实际上具有同样的含义，如 fish、fishes 和 fishing。将这些单词映射为共同的**概念性**类在一些场景下有用，这些场景包括对匹配单词的含义感兴趣，而不是对精确的拼写感兴趣。

这个映射过程称作**词干提取**（stemming），单词的词根形式也称作**词干**。一种常见的词干提取方法是**后缀去除**，即去除单词的结尾字母直到达到词根形式。一种非常经典且广泛使用的后缀去除法是 Porter Stemmer（一个后缀去除算法，M.F. Porter，1980）。

NLTK 提供了 Porter Stemmer 和其他词干提取的实现，如下所示。

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('fish')
'fish'
>>> stemmer.stem('fishes')
'fish'
```

```
>>> stemmer.stem('fishing')
'fish'
```

与大小写归一化类似，词干提取也是不可逆的。如果处理词干提取词，建议你保留一份原始文本的副本，以防需要重现。

与大小写归一化不同的是，词干提取是一个视语言而不同的任务。Snowball 是一个可选的词干提取器，支持多种语言，因此在处理多语言数据时非常有用。

```
>>> from nltk.stem import SnowballStemmer
>>> SnowballStemmer.languages
('danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian',
'italian', 'norwegian', 'porter', 'portuguese', 'romanian', 'russian',
'spanish', 'swedish')
>>> stemmer = SnowballStemmer('italian')
>>> stemmer.stem('pesce') # fish (noun)
'pesc
>>> stemmer.stem('pesci') # fishes
'pesc'
>>> stemmer.stem('pescare') # fishing
'pesc'
```

需要记住的最后一个细节是，词干只是单词的词根，并不总是真正的单词。

- **词形还原**

与词干提取类似，词形还原也对不同形式的单词进行分组。这样一来，可以将不同形式的单词当作同一个单词进行分析。

与词干提取不同的是，该过程需要一些额外知识，如需要进行词形还原的单词的正确词性标签。词形还原的输出称为**词目**（lemma），它是一个有效的单词。简单的后缀去除法对词形还原不起作用，因为一些不规则动词形式有完全不同的形态，如 go、goes、going 和 went 都应该映射为 go，但词干提取器对 went 的处理就出错了。

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('go')
'go'
>>> stemmer.stem('went')
'went'
```

NLTK 中的词形还原用 WordNet 实现。WordNet 是一个词汇资源，将英语单词分成同义词集合（也称为 `synsets`），并提供单词的定义和其他信息。

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize('go', pos='v')
'go'
>>> lemmatizer.lemmatize('went')
```

```
'went'
>>> lemmatizer.lemmatize('went', pos='v')
'go'
```

lemmatize()函数还接受一个可选参数，即词性标签。如果没有词性标签，那么词形还原器可能就会失败，如下所示。

```
>>> lemmatizer.lemmatize('am')
'am'
>>> lemmatizer.lemmatize('am', pos='v')
'be'
>>> lemmatizer.lemmatize('is')
'is'
>>> lemmatizer.lemmatize('is', pos='v')
'be'
>>> lemmatizer.lemmatize('are')
'are'
>>> lemmatizer.lemmatize('are', pos='v')
'be'
```

基于 WordNet 词形还原的可用词性标签可分为几类：形容词（a）、名词（n）、动词（v）和副词（r）。

- **停用词移除**

停用词是指并不承载内容的词，至少单独来看是这样，但它们在大多数自然语言中非常常见，如冠词和连词。信息检索领域的早期研究（如 Luhn，1958）显示，最有意义的单词并不是词频最高的词（也不是最低的词）。该研究的进一步发展是关于如何去除这些词而不丢失有意义的内容。在硬盘空间和内存有限且昂贵的年代，去除停用词可以大幅减少数据，在构建文档集合的索引时节约一些存储空间。

如今，太兆字节更加便宜，节约硬盘空间的动机比以前弱了，但去除**不感兴趣的**单词是否对特定应用有益仍然是一个问题。

找到去除停用词后对应用有害的反例非常容易。设想一个搜索引擎不对停用词进行索引，那么如何找到关于 The Who（一个英国摇滚乐队）或者莎士比亚著名台词“To be, or not to be: that is the question”的网页？乐队名称和台词中的所有词都是常见的英语停用词（除了 question）。不能去除停用词的例子数不胜数，现代搜索引擎（如 Google 和 Bing）都在索引中保留了停用词。

另一种做法是在频率分析中去除不感兴趣的词。通过观察一个单词的全局文档频率（有它出现的文档数量与总文档数量的比值），我们可以定义任意阈值以去除频率太高和太低的词。该方法也在 scikit-learn 库中实现了，其中不同的向量化构造器（如 TfidfVectorizer 类）都可以定义 max_df 和 min_df 参数。

在特定领域的集合中，一些词的频率会比在常用英语中更高。例如，在电影评论集合中，

movie 或 actor 可能会出现在几乎所有的文档中。虽然它们不是停用词（其实即使单独来看，这些词也都是有意义的），我们也可以将这些词当作停用词处理。只通过词频自动去除这些词也会带来额外的问题：如何获取 bad movie 或者 terrific actor 这样的信息？

经过上述讨论后，关于停用词移除的总结如下：停用词的移除或保留与应用高度相关。因此，不同应用需要对停用词移除这个预处理步骤做执行和不执行的测试，并比较效果。

网上有一些常见英语单词的清单可以用，因此自定义停用词列表非常简单。NLTK 也提供了自己的停用词列表，如下所示。

```
>>> from nltk.corpus import stopwords
>>> stop_list = stopwords.words('english')
>>> len(stop_list)
127
>>> stop_list[:10] # 前 10 个单词
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
'your']
```

- **同义词映射**

本节要讨论的最后一个归一化步骤是将一组同义词映射到同一项。其动机和大小写归一化非常相似，唯一的不同之处是，同义词并不是同一个单词。对这些词的映射可能是模糊的，因为这些词在不同的语境下表示不同的含义。

最简单的一种方法是使用受控词表，也就是提供映射关系的词汇资源。在 Python 中，它是字典形式的。

```
>>> synonyms = {'big': 'large', 'purchase': 'buy'}
```

在这个示例中，我们可以用 `synonyms` 将单词 `big` 翻译成 `large`，并将单词 `purchase` 翻译成 `buy`。该替换过程也非常简单，即获取字典中期望的键，例如，`synonyms['big']`将返回 `large`。一个字典还有 `get(key, default=None)`方法，其作用是获取特定的键。该方法需要第二个参数，若键不存在，则将第二个参数作为默认值。

```
>>> text = "I want to purchase a book on Big Data".lower().split()
>>> text
['i', 'want', 'to', 'purchase', 'a', 'book', 'on', 'big', 'data']
>>> new_text = [synonyms.get(word, word) for word in text]
>>> new_text
['i', 'want', 'to', 'buy', 'a', 'book', 'on', 'large', 'data']
```

以上代码用列表推导式迭代 `text` 列表。每个列表中的 `word` 用于从 `synonyms` 字典中检索一个潜在的同义词。如果没有可用的同义词，使用 `get()`的第二个参数将不会导致异常 `KeyError`，而是将原始单词作为输出。

处理自然语言时总会出现歧义问题。一方面，将 purchase 映射成 buy 似乎很合理，但从前面

的示例来看，large data 的含义是什么呢？问题是我们没有考虑单词 big 的使用语境，这里不应该将 big 孤立来看，而应该作为短语 Big Data 的一部分。除了这个特定的示例，还可以在自然语言中找到非常多一词多义的示例。

如前述 WordNet 这样的资源对于一个单词所有可能的语义分组提供了丰富的信息。为了强调这个问题并不能简单地用一个字典表示，思考以下示例。

```
>>> from nltk.corpus import wordnet
>>> syns = wordnet.synsets('big', pos='a')
>>> for syn in syns:
...   print(syn.lemma_names())
...
['large', 'big']
['big']
['bad', 'big']
['big']
['big', 'large', 'prominent']
['big', 'heavy']
['boastful', 'braggart', 'bragging', 'braggy', 'big', 'cock-a-hoop',
'crowing', 'self-aggrandizing', 'self-aggrandising']
['big', 'swelled', 'vainglorious']
['adult', 'big', 'full-grown', 'fully_grown', 'grown', 'grownup']
['big']
['big', 'large', 'magnanimous']
['big', 'bighearted', 'bounteous', 'bountiful', 'freehanded', 'handsome',
'giving', 'liberal', 'openhanded']
['big', 'enceinte', 'expectant', 'gravid', 'great', 'large', 'heavy',
'with_child']
```

我们可以看到，单词 big 用于多个同义词集合中。虽然它们或多或少是关于 big 这个概念的，但必须合理地捕捉其中的细微差别。

换句话说，这并不是一个简单的问题。单词语义消歧是一个非常活跃的研究领域，对一些有关语言的应用都有影响。特定数据领域的一些应用可能会受益于较小的受控词表，但在大部分常用英语的场景下，需要特别小心地处理同义词。

7

7.3.2 信息抽取

自然语言处理中较难但非常有趣的任务是从非结构化文本中抽取结构化信息。该过程通常叫作**信息抽取**，它包括一系列子任务，其中最流行的是**命名实体识别**（named entity recognition，NER）。

命名实体识别的目的是识别提到的实体。例如，识别一段文本中的人名或公司名称并为其分配正确的标签。常见的实体类型包括人名、机构名称、地点、数值、时间和货币。举个例子，查看以下文本：

"The Ford Motor Company (commonly referred to simply as Ford) is an American multinational automaker headquartered in Dearborn, Michigan, a suburb of Detroit. It was founded by Henry Ford and incorporated on June 16, 1903."

（从维基百科的福特公司页面获得的）以上片段包含一些命名实体。具体的总结如表 7-3 所示。

表 7-3　命名实体

实　　体	实体类型
Ford Motor Company	机构
Dearborn, Michigan	地点
Detroit	地点
Henry Ford	人物
June 16, 1903	时间

从抽取结构化信息的视角来看，这段文本实际上还描述了实体间的关系，如表 7-4 所示。

表 7-4　实体间的关系

主　　语	关　　系	宾　　语
Ford Motor Company	Located in	Detroit
Ford Motor Company	Founded by	Henry Ford

通常情况下，可以从文本中推断出一些额外关系。例如，如果福特是一家位于底特律的美国公司，那么我们可以推断出底特律在美国。



知识表示和推理

福特的例子只是一个简化的示例。需要注意的是，知识表示是一个复杂的主题，并且本身是一个研究领域，它涉及多个学科，其中包括人工智能、信息检索、自然语言处理、数据库和语义网。

现在有很多知识表示的标准（如 RDF 格式），在设计知识库的结构时，理解应用所处的商业领域以及如何使用这些知识非常重要。

信息抽取系统的大致架构如图 7-3 所示。

上述信息抽取系统的第一个模块是**预处理**，但它实际上包含了 7.3.1 节中介绍的多个步骤。原始输入通常是纯文本，因此是一个字符串。该假设意味着我们已经从相关格式（如 JSON 文件、数据库等）中抽取了文本。预处理的输出结果取决于所选择的步骤。为了抽取实体，没有执行停用词移除、词干提取和归一化等步骤，因为这样会改变原始单词的序列，并可能妨碍多项实体的识别。例如，如果将 Bank of America 转换为 bank america，那么不可能正确地识别实体。这里只会进行分词和词性标注，因此，输出是一个元组列表`(term, pos_tag)`。

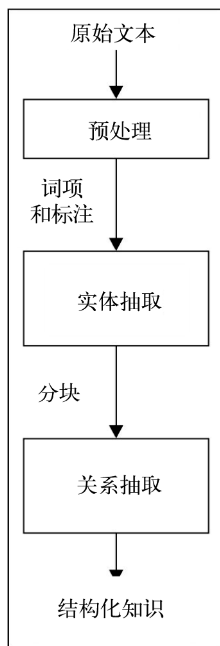


图 7-3 信息抽取系统的概览

流水线的第二个步骤是**识别实体**。从一组词项的列表出发，实体被分割并根据其类型进行标注。对于不同的应用，我们可能只对普通名词感兴趣，也可能希望包括无定名词或名词词组。该步骤的输出是一个块列表，这些块是短小且不重叠的短语，并且是以树的形式组织的。分块也称为浅解析或者部分解析。与此概念相对的是完全解析，目的是创建一个完全解析树。

最后一步是**识别实体间的关系**。关系非常有趣，它允许我们推断文本中的实体。由于语言处理的困难性，这一步通常是最难的。它也取决于前一阶段的输出质量。例如，如果实体抽取的结果缺失某个实体，那么就会缺失所有对应的相关关系。最后的输出可以由一个元组`(subject, relation, object)`列表表示，福特公司的示例中介绍过该格式，也可以根据应用的需要定义为其他格式。

NLTK 开箱即用的实现允许我们轻松地执行命名实体抽取。`nltk.chunk.ne_chunk()`函数将`(term, pos_tag)`格式的一个元组列表作为输入，并返回一个解析树。如果正确识别，每个命名实体都是与对应的实体类型（如人名、位置等）相关联的。该函数还有第二个可选参数`binary`，默认为 `False`。当该参数为 `True` 时，该函数会取消实体类型识别，而且实体只会被简单地标注为 `NE`（命名实体）。

以下示例将实现以上步骤并用维基百科 API 获取已识别实体的简短摘要。

首先定义库的导入和 `ArgumentParser`。

```
# Chap07/blogs_entities.py
from argparse import ArgumentParser
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.chunk import ne_chunk
import wikipedia

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--entity')
  return parser
```

然后定义一个函数来迭代进行过词性标注的词项列表，并返回一个名词短语列表。名词短语是标注了有关名词标签（如 NN 或 NNP）的词项序列。可以将该名词列表与 NLTK 识别的实际实体进行比较。

```
def get_noun_phrases(pos_tagged_tokens):
  all_nouns = []
  previous_pos = None
  current_chunk = []
  for (token, pos) in pos_tagged_tokens:
    if pos.startswith('NN'):
      if pos == previous_pos:
        current_chunk.append(token)
      else:
        if current_chunk:
          all_nouns.append((' '.join(current_chunk),
                           previous_pos))
        current_chunk = [token]
    else:
      if current_chunk:
        all_nouns.append((' '.join(current_chunk),
                         previous_pos))
        current_chunk = []
    previous_pos = pos
  if current_chunk:
    all_nouns.append((' '.join(current_chunk), pos))
  return all_nouns
```

接着定义一个接收解析树和实体类型的函数，并根据特定的实体类型从给定的树返回实体列表。

```
def get_entities(tree, entity_type):
  for ne in tree.subtrees():
    if ne.label() == entity_type:
      tokens = [t[0] for t in ne.leaves()]
      yield ' '.join(tokens)
```

最后，脚本的核心逻辑是：从维基百科获取特定的实体，检索它的简短摘要，然后对摘要进行分词和词性标注。从简短的摘要中识别出名词短语和命名实体。每个实体会与来自维基百科的摘要同时显示。

```
if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  entity = wikipedia.summary(args.entity, sentences=2)

  tokens = word_tokenize(entity)
  tagged_tokens = pos_tag(tokens)
  chunks = ne_chunk(tagged_tokens, binary=True)

  print("-----")
  print("Description of {}".format(args.entity))
  print(entity)
  print("-----")
  print("Noun phrases in description:")
  for noun in get_noun_phrases(tagged_tokens):
    print(noun[0])  # tuple (noun, pos_tag)
  print("-----")
  print("Named entities in description:")
  for ne in get_entities(chunks, entity_type='NE'):
    summary = wikipedia.summary(ne, sentences=1)
    print("{}: {}".format(ne, summary))
```

可以用以下命令执行以上脚本。

```
$ python blogs_entities.py --entity London
```

代码的部分输出如图 7-4 所示。

```
-----
Description of London
London /ˈlʌndən/ is the capital and most populous city of England and
the United Kingdom. Standing on the River Thames in the south east of
Great Britain, London has been a major settlement for two millennia.
-----
Noun phrases in description:
London
/ˈlʌndən/
capital
# more nouns ...
-----
Named entities in description:
London: London /ˈlʌndən/ is the capital and most populous city of
England and the United Kingdom.
England: England /ˈɪŋɡlənd/ is a country that is part of the United #
# more entities ...
```

图 7-4　代码的输出

识别出的完整的名词短语如图 7-5 所示。

```
London, /ˈlʌndən/, capital, city, England, United Kingdom, Standing,
River Thames, south east, Great Britain, London, settlement, and
millennia.
```

图 7-5　识别出的名词短语

在这个示例中，只有 `Standing` 被错误地标注为名词，可能是因为其首字母大写的 S，但这只是因为它出现在句子的开头。虽然不一定所有词都是我们感兴趣的（如 `Standing`、`settlement`、`millennia`），但所有名词都被识别出了。另一方面，识别的命名实体列表看起来非常棒：`London`、`England`、`United Kingdom`、`River Thames`、`Great Britain` 和 `London`。

7.4　小结

本章介绍了自然语言处理，这是一个充满挑战和机遇的复杂领域。

本章的第一部分重点介绍了如何从 Web 上获取文本数据。博客是文本挖掘的天然候选对象，因为其中充满了丰富的文本数据。处理了两个最流行的免费博客平台（WordPress.com 和 Blogger）后，我们将数据获取的问题统一为如何从网站订阅源的 XML 解析获得数据，其中特别介绍了两个订阅格式 RSS 和 Atom。基于在网站中的重要性以及在网络用户每日生活中的重要程度，维基百科也是非常重要的文本数据获取源。我们介绍了如何用 Python 与这些服务交互，实现途径是通过 Python 中可用的库或者快速编写我们自己的一些函数。

本章的第二部分是关于自然语言处理的。本书前面已经介绍过一些自然语言处理的概念，但这里是首次用更系统和正式的方式介绍。我们介绍了自然语言处理流水线，该流水线可以实现从原始文本到命名实体识别的所有必需预处理步骤，从而使得高级的应用成为可能。

下一章将重点介绍其他社会媒体 API，为更广泛的数据挖掘提供可能。

第 8 章

挖掘所有数据

本章将介绍一些可用的社会媒体 API。我们将讨论以下主题：

- 如何从 YouTube 挖掘视频
- 如何从 GitHub 挖掘开源项目
- 如何从 Yelp 挖掘本地商家
- 如何用 requests 库调用任何 Web API
- 如何将你的 requests 请求包装到一个自定义的客户端

8.1 很多社交 API

前面的每一章都重点介绍了一种近几年特别流行的社会媒体平台。但故事还远未结束。很多平台都提供了社交网络功能，以及用于挖掘数据的非常好的 API。不过，详尽描述所有可能的 API 及其相应示例超出了本书的范围。

为了帮助你进一步思考，本章解决社会媒体挖掘的两个方面。首先，我们将介绍一些非常有趣的 API 以便搜索和挖掘一些复杂的实体，如视频、开源项目或本地商家。其次，我们将介绍特定的 API 没有可用的 Python 客户端时应该如何做。

8.2 挖掘 YouTube 上的视频

YouTube 可能不需要过多介绍，它是世界上访问次数最多的网站之一（2016 年 3 月的 Alexa 排名中位列第二）。该平台上分享的视频服务包括的内容非常广泛，并且这些内容由广泛的作者群体（从业余视频博主到商业公司）生产。YouTube 的注册用户可以上传视频、对视频评分并评论视频。查看和分享并不需要用户注册。

YouTube 于 2006 年被 Google 收购，因此如今它是 Google 平台的一部分。我们已经在第 5 章和第 7 章中介绍过 Google 的其他服务，特别是 Google+和 Blogger。YouTube 提供了三种 API

帮助你将自己的应用整合到 YouTube：YouTube 数据、YouTube 分析和 YouTube 报告。我们将重点介绍第一种，并介绍如何从 YouTube 检索和挖掘数据，而其他两种都是为内容生产者提供的。

接入 YouTube 数据 API 的第一步与我们前面看到的非常类似，因此你可以回顾第 5 章来查看如何接入 Google Developers Console。如果已经在 Google Developers Console 注册了凭证，你可以打开 YouTube 数据 API，如图 8-1 所示。

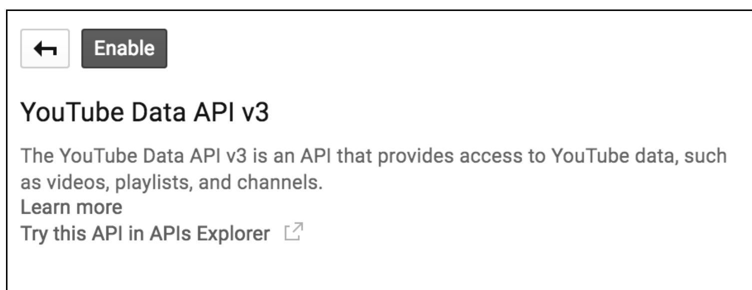


图 8-1　从 Google Developers Console 打开 YouTube 数据 API

通过 Google Developers Console 的 **Credentials** 选项卡，我们可以重复使用第 5 章中所创建 Google+项目的 API 密钥。解决好凭证后，可以将我们的 Google API 密钥作为一个环境变量。

```
$ export GOOGLE_API_KEY="your-api-key"
```

我们将重用用于其他 Google 服务的 Python 客户端，可以通过 CheeseShop 安装该客户端。

```
$ pip install google-api-python-client
```

提醒一句，以上安装的包也可以用 `googleapiclient` 的名称，以 `apiclient` 作为别名（在示例代码中使用）。

该 API 的接入限制使用了一个配额系统。每个应用每天的访问量上限是 1 000 000 次。对不同端点的不同调用也有不同的配额，但是 1 000 000 的频率限制对于我们做实验来说是充分够用的。具体的配额和使用可以在 Google Developers Console 中查看。

与 YouTube 数据 API 交互的第一个示例是搜索应用。`youtube_search_video.py` 脚本实现了对 `search.list` API 端点的调用。该端点的访问配额是 100 次。

```
# Chap08/youtube_search_video.py
import os
import json
from datetime import datetime
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
```

```
  parser = ArgumentParser()
  parser.add_argument('--query')
  parser.add_argument('--n', type=int)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  api_key = os.environ.get('GOOGLE_API_KEY')
  service = build('youtube',
                  'v3',
                  developerKey=api_key)

  search_feed = service.search()
  search_query = search_feed.list(q=args.query,
                                  part="id,snippet",
                                  maxResults=args.n,
                                  type='channel')
  search_response = search_query.execute()

  print(json.dumps(search_response, indent=4))
```

以上脚本用 `ArgumentParser` 方法解析命令行参数。`--query` 参数允许我们向 API 调用传递查询，而`--n` 参数（可选，默认为 `10`）用于定义我们希望检索的结果的数量。

与 YouTube API 交互的核心由 `service` 对象处理，该对象是通过调用 `apiclient.discovery.build()`函数实例化的，这与 Google+ API 和 Blogger API 的交互类似。该函数的两个可选参数是服务名称（`youtube`）和 API 的版本（`v3`），第三个关键字参数是 API 密钥，我们已经将其定义为环境变量。实例化 `service` 对象后，可以调用其 `search()`函数来构建搜索源对象。根据实际 API 请求的定义，该对象允许我们调用 `list()`函数。

`list()`函数只以关键字作为输入参数。`q` 参数是我们传递给 API 的查询。`part` 参数允许我们定义一个以逗号分隔的属性列表，其中应该包括 API 响应。最后，我们可以执行对 API 的调用，这会产生一个响应对象（即一个 Python 字典）。为了简洁起见，用 `json.dumps()`函数漂亮地打印它。

`search_response` 字典有几个第一层级的属性，如下所示。

- `pageInfo`，包括 `resultsPerPage` 和 `totalResults`
- `items`：搜索结果列表
- `kind`：结果对象的类型（本例中为 `youtube#searchListResponse`）
- `etag`
- `regionCode`：两个字母的国家代码，如 `GB`
- `nextPageToken`：一个令牌，用于创建下一个页面结果的调用

在 `items` 列表中，每个项是一个搜索结果。我们有项（一个视频、频道或播放列表）的 ID

以及一段包含细节的描述。

有关视频的一些有趣细节如下所示。

- channelId：视频创建者的 ID
- channelTitle：视频创建者的名字
- title：视频的标题
- description：视频的文本描述
- publishedAt：ISO 8601 格式的发布日期字符串

可以扩展基础查询来自定义搜索请求的结果。例如，来自脚本 youtube_search_video.py 中的查询检索的不只是视频，还包括播放列表或频道。如果想要将搜索结果限定到特定对象，可以使用 list() 函数中的 type 参数，它以 video、playlist 或 channel 作为值。此外，我们可以用 order 属性影响结果的排序。该属性可以选择以下其中之一。

- date：按时间逆序排列（最新的排在最前面）
- rating：将结果按评分从高到低排列
- relevance：排序的默认选项，根据与查询语句的相关性排序
- title：根据标题的字母顺序排列
- videoCount：根据上传视频的数量对频道降序排列
- viewCount：根据观看量对视频从高到低排列

最后，可以用 publishedBefore 和 publishedAfter 参数将搜索结果限制到特定的发布日期。

我们在以下示例中实践以上信息，并构建一个搜索 API 端点的自定义查询。假设我们想检索发布于 2016 年 1 月的视频，并希望按照相关性对检索结果排序，那么可以如下重构对 list() 函数的调用。

```
search_query = search_feed.list(
                                 q=args.query,
                                 part="id,snippet",
                                 maxResults=args.n,
                                 type='video',
                                 publishedAfter='2016-01-01T00:00:00Z',
                                 publishedBefore='2016-02-01T00:00:00Z')
```

publishedBefore 和 publishedAfter 参数期望的日期和时间格式是 RFC 3339。

为了处理更多任意数量的搜索结果，我们需要实现一个翻页机制。最简单的方法是将对搜索端点的调用包进一个自定义函数中，并迭代不同的页面，直到获得期望的结果。

以下类实现了一个自定义的 YouTube 客户端，该客户端使用的方法仅用于视频搜索。

```
class YoutubeClient(object):

  def __init__(self, api_key):
    self.service = build('youtube',
                         'v3',
                         developerKey=api_key)

  def search_video(self, query, n_results):
    search = self.service.search()
    request = search.list(q=query,
                          part="id,snippet",
                          maxResults=n_results,
                          type='video')
    all_results = []
    while request and len(all_results) <= n_results:
      response = request.execute()
      try:
        for video in response['items']:
          all_results.append(video)
      except KeyError:
        break
      request = search.list_next(request, response)
    return all_results[:n_results]
```

在初始化过程中，YoutubeClient 类需要用 api_key 来调用 apiclient.discovery.build()方法并创建服务。

搜索逻辑的核心是在 search_video()方法中实现的，它需要两个参数：查询，以及期望的结果数量。我们先用 list()方法建立 request 对象，和之前一样。while 循环检查 request 对象是否为 None，以及我们是否达到了期望的结果数量。查询的运行会检索 response 对象中的结果，该对象是一个字典。关于视频的数据列在 response['items']中，并追加到了 all_results 列表中。循环的最后一个步骤是调用 list_next()方法，该方法会覆盖 request 对象，为下一个页面的结果做准备。

YoutubeClient 类的应用如下所示。

```
# Chap08/youtube_search_video_pagination.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--query')
  parser.add_argument('--n', default=50, type=int)
  parser.add_argument('--output')
  return parser

class YoutubeClient(object):
```

```
  # 与前面的代码段定义的相同

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  api_key = os.environ.get('GOOGLE_API_KEY')

  youtube = YoutubeClient(api_key)
  videos = youtube.search_video(args.query, args.n)

  with open(args.output, 'w') as f:
    for video in videos:
      f.write(json.dumps(video)+"\n")
```

可以用以下方式调用以上脚本。

```
$ python youtube_search_video_pagination.py \
  --query python \
  --n 50 \
  --output videos.jsonl
```

执行以上命令将生成 videos.jsonl 文件，包含 50 个与查询 `python` 相关的视频。该文件以 JSON Lines 格式保存。

本节给出了与 YouTube 数据 API 交互的示例。该模式与我们接触过的 Google+和 Blogger 非常相似，因此，一旦理解一种 Google 服务的交互方法，就可以很容易地复制到其他 Google 服务。

8.3 挖掘 GitHub 上的开源软件

GitHub 是一个提供 Git 仓库的托管服务。虽然其中的私有仓库需要付费，但 GitHub 服务的知名在于对开源服务的托管。除了由 Git 提供的源控制管理功能之外，GitHub 还提供了使管理开源项目更加容易的很多功能（如错误追踪、维基、特征请求等）。



源控制管理软件

源控制系统（也称作版本控制或代码修改控制）是软件管理中最重要的工具之一，它追踪了软件在开发中的演进过程。这一方面经常被新手或独立开发者所忽视，但若以团队或个人的形式完成复杂项目，版本控制至关重要。它的优点很多，其中包括提供了对不需要的更改进行回退的可能，并提供了与团队成员高效协作的可能。Git 最开始是由 Linux 之父林纳斯·托瓦兹开发的，是最流行的版本控制工具。了解版本控制系统的基础对新手开发者、分析师和研究人员大有益处。

GitHub 提供了通过一个 API（https://developer.github.com/v3/）接入其数据的途径，使用的是在其平台上注册应用的常见机制。只在获取鉴权用户的私人信息时才需要鉴权应用。

该 API 有一些严格的频率限制。非鉴权的请求限制为 60 次/小时，这是一个较低的数值。完成鉴权后，频率限制将提升到 5000 次/小时。搜索 API 也有自定义的接口访问频率限制（未鉴权为 10 次/分钟，鉴权为 30 次/分钟）。可以通过以下三种方式进行鉴权：

- 基本的用户名/密码鉴权
- 通过 OAuth2 令牌
- 通过 OAuth2 客户端 ID 和客户端密码

最基本的鉴权需要将实际的用户名和密码传到 API 端点。鉴权是通过 OAuth2 令牌完成的，也就是说，它需要以程序的方式获取该令牌，然后通过头文件或 URL 参数发送给 API 端点。最后的选项是将客户端 ID 和客户端密码传递给 API。注册 GitHub 平台的应用时可以获得这些细节。图 8-2 展示了创建新应用的注册界面。

图 8-2　在 GitHub 上注册一个新的应用

注册好应用后，可以将凭证保存为环境变量。

```
$ export GITHUB_CLIENT_ID="your-client-id"
$ export GITHUB_CLIENT_SECRET="your-client-secret"
```

配置好以上步骤后，就可以开始与 GitHub API 交互了。Python 提供了多个客户端，这些库都是第三方的，并不是 GitHub 官方支持的。本节选择的是 PyGithub，但如果感兴趣，你也可以选择其他库进行尝试。

用以下命令将该库安装到虚拟环境中。

```
$ pip install PyGithub
```

以下脚本的作用是查找特定的用户名，以获得用户的基本信息及其 GitHub 仓库。

```
# Chap08/github_get_user.py
import os
from argparse import ArgumentParser
from github import Github
from github.GithubException import UnknownObjectException

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--user')
  parser.add_argument('--get-repos', action='store_true',
                      default=False)
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  client_id = os.environ['GITHUB_CLIENT_ID']
  client_secret = os.environ['GITHUB_CLIENT_SECRET']

  g = Github(client_id=client_id, client_secret=client_secret)

  try:
    user = g.get_user(args.user)
    print("Username: {}".format(args.user))
    print("Full name: {}".format(user.name))
    print("Location: {}".format(user.location))
    print("Number of repos: {}".format(user.public_repos))
    if args.get_repos:
      repos = user.get_repos()
      for repo in repos:
        print("Repo: {} ({} stars)".format(repo.name,
              repo.stargazers_count))
  except UnknownObjectException:
    print("User not found")
```

该脚本用 `ArgumentParser` 从命令行获取参数：`--user` 用于传递待搜索的用户名，可选参数`--get-repos` 是一个布尔标记，表明我们是否希望在输出中包括用户仓库的列表（为简便起见，这里没有涉及翻页的情况）。

可以用以下命令执行上述脚本。

```
$ python github_get_user.py --user bonzanini --get-repos
```

以上命令会产生以下输出。

```
Username: bonzanini
Full name: Marco Bonzanini
Location: London, UK
Number of repos: 9
Repo: bonzanini.github.io (1 stars)
Repo: Book-SocialMediaMiningPython (3 stars)
# 更多仓库……
```

`Github.get_user()`函数假设我们已经知道了要寻找的用户名称。

用户对象的一些有趣属性如表 8-1 所示。

表 8-1 用户对象的属性

属性名称	描　述
avatar_url	头像的 URL
bio	用户的简短介绍
blog	用户的博客
company	用户的公司
created_at	资料的创建时间
email	用户的电子邮件
followers	用户的粉丝数量
followers_url	用户粉丝列表的 URL
following	用户关注的资料数量
following_url	用户关注的资料列表的 URL
location	用户的地理位置
login	登录名
name	全名
public_repos	公开仓库的数量
public_gists	公开 gist 的数量
repos_url	检索仓库的 URL

用准确的用户名检索用户资料后，我们来看看如何检索用户。GitHub API 通过 `Github.search_users()`函数提供了端点，该函数允许我们指定查询语句、排序和顺序参数。以下脚本实现了搜索。


```
# Chap08/github_search_user.py
import os
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from github import Github

def get_parser():
```

```
  parser = ArgumentParser()
  parser.add_argument('--query')
  parser.add_argument('--sort',
                      default='followers',
                      type=check_sort_value)
  parser.add_argument('--order',
                      default='desc',
                      type=check_order_value)
  parser.add_argument('--n', default=5, type=int)
  return parser

def check_sort_value(value):
  valid_sort_values = ['followers', 'joined', 'repositories']
  if value not in valid_sort_values:
     raise ArgumentTypeError('"{}" is an invalid value for
                             "sort"'.format(value))
  return value

def check_order_value(value):
  valid_order_values = ['asc', 'desc']
  if value not in valid_order_values:
     raise ArgumentTypeError('"{}" is an invalid value for
                             "order"'.format(value))
  return value

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  client_id = os.environ['GITHUB_CLIENT_ID']
  client_secret = os.environ['GITHUB_CLIENT_SECRET']

  g = Github(client_id=client_id, client_secret=client_secret)

  users = g.search_users(args.query,
                         sort=args.sort,
                         order=args.order)
  for i, u in enumerate(users[:args.n]):
    print("{}) {} ({}) with {} repos ".format(i+1, u.login,
          u.name, u.public_repos))
```

该脚本用 `ArgumentParser` 从命令行解析参数。可以通过 `--query` 选项传递搜索语句。该 API 还接受两个可选参数来自定义排序和顺序。这两个值分别用 `--sort` 参数和 `--order` 参数表示，只接受特定的值，因此我们必须分别为这两个参数指定类型。更准确地说，`check_sort_value()` 和 `check_order_value()` 辅助函数实现了以下逻辑：如果给出的是一个合法值，它会被传递给 API，否则 `ArgumentTypeError` 会抛出异常。最后，`ArgumentParser` 还接受一个 `--n` 参数，以指定期望的结果数量（默认为 5）。

可以用以下命令运行上述脚本。

```
$ python github_search_user.py \
  --query [your query here] \
```

```
  --sort followers \
  --order desc
```

输出是 5 个（默认值）最受欢迎用户的列表，即粉丝数量最多的用户。

有个类似的脚本可用于搜索受欢迎的仓库。具体实现如下所示。

```
# Chap08/github_search_repos.py
import os
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from github import Github

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--query')
  parser.add_argument('--sort',
                      default='stars',
                      type=check_sort_value)
  parser.add_argument('--order',
                      default='desc',
                      type=check_order_value)
  parser.add_argument('--n', default=5, type=int)
  return parser

def check_sort_value(value):
  valid_sort_values = ['stars', 'forks', 'updated']
  if value not in valid_sort_values:
     raise ArgumentTypeError('"{}" is an invalid value for
                             "sort"'.format(value))
  return value

def check_order_value(value):
  valid_order_values = ['asc', 'desc']
  if value not in valid_order_values:
     raise ArgumentTypeError('"{}" is an invalid value for
                             "order"'.format(value))
  return value

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  client_id = os.environ['GITHUB_CLIENT_ID']
  client_secret = os.environ['GITHUB_CLIENT_SECRET']

  g = Github(client_id=client_id, client_secret=client_secret)

  repos = g.search_repositories(args.query,
                                sort=args.sort,
                                order=args.order)
  for i, r in enumerate(repos[:args.n]):
    print("{}) {} by {} ({} stars) Fullname: {}".format(i+1, r.name, r.owner.name,
          r.stargazers_count, r.full_name))
```

上述 `github_search_repos.py` 脚本与前面的代码很相似，并且也使用了带有`--sort` 和

--order 参数的 ArgumentParser。为了搜索仓库,这里使用了 Github.search_repositories() 方法，该方法以一个查询语句和一些关键字参数作为输入。

用以下命令运行上述脚本。

```
$ python github_search_repos.py \
  --query python \
  --sort stars \
  --order desc
```

用 python 查询语句得到了一些有趣的结果。

```
1) oh-my-zsh by Robby Russell (36163 stars)
2) jQuery-File-Upload by Sebastian Tschan (23381 stars)
3) awesome-python by Vinta (20093 stars)
4) requests by Kenneth Reitz (18616 stars)
5) scrapy by Scrapy project (13652 stars)
```

虽然以上列表中出现了一些流行的 Python 库（如 requests 和 Scrapy），但列表的前两个结果似乎与 Python 没有关系。这是因为调用搜索 API 时默认搜索标题/描述字段，所以该查询会得到一些只在描述中提到了 python 的结果，但实际上不是用 Python 实现的。

我们可以重构搜索以获得特定语言实现的仓库。在上面的代码中，我们只需要用 language 量化器修改 Github.search_repositories()的调用，如下所示。

```
repos = g.search_repositories("language:{}".format(args.query),
                              sort=args.sort,
                              order=args.order)
```

对查询进行小改动后，结果如下所示。

```
1) httpie by Jakub RoztoÄil (22130 stars)
2) awesome-python by Vinta (20093 stars)
3) thef*** by Vladimir Iakovlev (19868 stars)
4) flask by The Pallets Projects (19824 stars)
5) django by Django (19044 stars)
```

此时得到的所有结果基本上都是由 Python 实现的项目。

表 8-2 总结了仓库对象的主要属性。

表 8-2　仓库对象的属性

属性名称	描　　述
name	出现在 URL 中的仓库名称
full_name	仓库的完整名称（即 user_name/repo_name）
owner	表示拥有项目的用户对象
id	仓库的数值 ID

（续）

属性名称	描　述
stargazers_count	获得的星星数量
description	仓库的文本描述
created_at	创建时间的 datetime 对象
updated_at	最近更新时间的 datetime 对象
open_issues_count	开放问题的数量
language	仓库的主要语言
languages_url	检索其他语言信息的 URL
homepage	项目主页的 URL

还有关于特定仓库的很多属性和细节。API 的文档（https://developer.github.com/v3/repos/）提供了具体的描述。

本节展示了从 GitHub API 检索数据的示例。潜在的应用包括分析并发现最常用的编程语言，如哪一种语言的项目最多、哪一种语言的项目最活跃、哪一种语言的项目在某时间段的开放问题最多等。

8.4　挖掘 Yelp 上的本地商家

Yelp 是一个在线服务，用于托管本地商家众包点评[①]，比如酒吧和餐馆。它的月活跃用户数为 135 000 000，内容主要由社区驱动。

Yelp 提供了三个 API 来搜索本地商家数据并与之交互。这三个 API 按照目的进行分组。搜索 API 是基于关键词的搜索的接入点。商家 API 用于查找关于特定商家的信息。电话搜索 API 用于通过电话号码查找商家。

Yelp API 的接口访问频率限制为每日 25 000 次。

本节描述了建立应用并用特定关键词搜索 Yelp 数据库的一些基本步骤。

API 的接入由令牌保护。在 Yelp 注册账号后，接入任何 API 的第一步都是在开发者网站获得令牌。图 8-3 显示了接入 API 需要填写的信息。在这个示例中，Your website URL 字段指向本地主机。

① 类似于大众点评网。——译者注

Complete the following form to gain access to the Yelp API.

Your website URL:

http://127.0.0.1:5000

How you will use the API:

Testing Yelp API capabilities

Industry:

Other

Publishing

图 8-3　接入 Yelp API

完成第一步后，我们将获得四个接入键。应该将这些字符串保护起来并永不共享。接入键也称为客户键、客户密码、令牌和令牌密码。这些字符串用于调用 API 之前的 OAuth 鉴权过程。

按照本书惯例，我们将这些配置保存在环境变量中。

```
$ export YELP_CONSUMER_KEY="your-consumer-key"
$ export YELP_CONSUMER_SECRET="your-consumer-secret"
$ export YELP_TOKEN="your-token"
$ export YELP_TOKEN_SECRET="your-token-secret"
```

为了以程序的方式接入 API，我们需要安装官方的 Python 客户端。用以下命令将其安装到虚拟环境中。

```
$ pip install yelp
```

以下脚本 `yelp_client.py` 定义了一个函数，我们会用该函数创建对 API 的调用。

```
# Chap08/yelp_client.py
import os
from yelp.client import Client
from yelp.oauth1_authenticator import Oauth1Authenticator

def get_yelp_client():
  auth = Oauth1Authenticator(
    consumer_key=os.environ['YELP_CONSUMER_KEY'],
    consumer_secret=os.environ['YELP_CONSUMER_SECRET'],
    token=os.environ['YELP_TOKEN'],
    token_secret=os.environ['YELP_TOKEN_SECRET']
    )

  client = Client(auth)
  return client
```

一切准备就绪后，就可以开始搜索本地商家了。`yelp_search_business.py` 通过提供的地点和关键词搜索 API。

```
# Chap08/yelp_search_business.py
from argparse import ArgumentParser
from yelp_client import get_yelp_client

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--location')
  parser.add_argument('--search')
  parser.add_argument('--language', default='en')
  parser.add_argument('--limit', default=20)
  parser.add_argument('--sort', default=2)
  return parser

if __name__ == '__main__':
  client = get_yelp_client()
  parser = get_parser()
  args = parser.parse_args()

  params = {
    'term': args.search,
    'lang': args.language,
    'limit': args.limit,
    'sort': args.sort
  }

  response = client.search(args.location, **params)
  for business in response.businesses:
    address = ', '.join(business.location.address)
    categories = ', '.join([cat[0] for cat in
                            business.categories])
    print("{} id={} ({}, {}); rated {}; categories {}".format(business.name,
          business.id, address, business.location.postal_code,
          business.rating, categories))
```

ArgumentParser 对象用于解析来自命令行的参数。有两个参数是必需的：--location 和--search，前者是城市或本地商圈的名字，后者允许我们为某个搜索传递特定的参数。可选参数--language 用于指明输出的期望语言（特别是对于评论），它的值必须是一个语言代码（例如，英语是 en、法语是 fr、德语是 de 等）。我们还提供了另外两个可选参数：--limit 是我们期望的结果数量（默认为 20），--sort 则定义了结果排序的方式（0 表示最佳匹配、1 表示按距离排序、2 表示按星级排序）。

在脚本的主要部分，我们创建了 Yelp 客户端和解析器。然后定义了一个传递给 Yelp 客户端的参数字典。这些参数包括通过命令行定义的参数。

可以用以下命令调用脚本。


```
$ python yelp_search_business.py \
  --location London \
  --search breakfast \
  --limit 5
```

输出结果是打印到控制台的本地商家（这里是 5 家）列表。

```
Friends of Ours (61 Pitfield Street, N1 6BU); rated 4.5; categories Cafes,
Breakfast & Brunch, Coffee & Tea
Maison D'être (154 Canonbury Road, N1 2UP); rated 4.5; categories Coffee &
Tea, Breakfast & Brunch
Dishoom (5 Stable Street, N1C 4AB); rated 4.5; categories Indian, Breakfast
& Brunch
E Pellicci (332 Bethnal Green Road, E2 0AG); rated 4.5; categories Italian,
Cafes, Breakfast & Brunch
Alchemy (8 Ludgate Broadway, EC4V 6DU); rated 4.5; categories Coffee & Tea,Cafes
```

当搜索多个关键词时，查询必须用双引号包裹起来以便被正确地解析，如下所示。

```
$ python yelp_search_business.py \
  --location "San Francisco" \
  --search "craft beer" \
  --limit 5
```

API 返回的 `location` 对象具有一些属性，表 8-3 总结了一些最有趣的属性。

表 8-3　`location` 对象的属性

属性名称	描　述
`id`	商家的标识符
`name`	商家的名称
`image_url`	商家照片的 URL
`is_claimed`	布尔属性值，表示商家是否认领了该页面
`is_closed`	布尔属性值，表示该商家是否永久关闭
`url`	商家的 Yelp URL
`mobile_url`	商家的 Yelp 移动端 URL
`phone`	国际格式的电话号码字符串
`display_phone`	用于显示的电话号码字符串
`categories`	与商家相关的类别列表。每个类别是一个元组(`display_name`, `filter_name`)，这里的第一个名字用于显示，第二个名字用于过滤查询
`review_count`	有关商家的评论数量
`rating`	有关商家的评分（如 1、1.5……4.5、5）
`snippet_text`	商家的简短描述
`snippet_image_url`	与商家相关的图像的 URL
`location`	与商家相关的 `location` 对象

`location` 对象提供了以下属性。

- `address (list)`只包含地址字段
- `display_address (list)`是用于显示的格式，包括交叉路口、城市、州代码等
- `city (string)`

- state_code (string)是 ISO 3166-2 格式的州代码
- postal_code (string)
- country_code (string)是 ISO 3166-1 格式的国家代码
- cross_streets (string)是商家的交叉路口
- neighborhoods (list)是商家的周边信息
- coordinates.latitude (number)
- coordinates.longitude (number)

除了搜索 API，Yelp 还提供了一个商家 API 来专门检索特定商家的信息。该 API 假设你已经具有商家的特定 ID。

以下脚本展示了如何用 Python 客户端接入商家 API。

```
# Chap08/yelp_get_business.py
from argparse import ArgumentParser
from yelp_client import get_yelp_client

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--id')
  parser.add_argument('--language', default='en')
  return parser

if __name__ == '__main__':
  client = get_yelp_client()
  parser = get_parser()
  args = parser.parse_args()

  params = {
    'lang': args.language
  }

  response = client.get_business(args.id, **params)
  business = response.business
  print("Review count: {}".format(business.review_count))
  for review in business.reviews:
    print("{} (by {})".format(review.excerpt, review.user.name))
```

商家 API 的一个主要限制是，并不是完全提供给定商家的评论列表，它实际上只提供一个评论。这单个评论是商家 API 提供而搜索 API 没有展示的唯一信息，因此，商家 API 并不能提供太多数据挖掘空间。

有意思的是，Yelp 提供了一个随着时间不断更新和丰富的数据集，主要用于学术研究。该数据集包含商家、用户、评论、图片和用户间的连接等细节信息。该数据集可以用于自然语言处理、图挖掘和一般的机器学习。该数据集的数据挖掘挑战对学生来说是技术能力的证明。

8.5　创建自定义的 Python 客户端

本书通过 Python 库接入了不同的社会媒体平台，这些库要么是社会媒体官方支持的，要么是第三方提供的。

本节将回答以下问题：如果社会媒体平台不提供 Python 客户端及相应 API（也没有非官方库），那么该怎么办？

HTTP 让事情变得简单

为了实现对特定 API 的调用，我们推荐的 HTTP 交互库是 requests。可以用以下命令将其安装到虚拟环境中。

```
$ pip install requests
```

requests 库提供了非常直接的接口来执行 HTTP 调用。为了测试该库，我们将为 httpbin.org 服务实现一个简单的客户端。httpbin.org 服务是一个 Web 服务，提供了基本的请求/响应交互，能够满足我们的学习目的。

httpbin.org 的页面（http://httpbin.org/）解释了各种端点及含义。为了演示 requests 库，我们将实现一个 HttpBinClient 类，其中包含一些基本端点的调用方法。

```
class HttpBinClient(object):

  base_url = 'http://httpbin.org'

  def get_ip(self):
    response = requests.get("{}/ip".format(self.base_url))
    my_ip = response.json()['origin']
    return my_ip

  def get_user_agent(self):
    response = requests.get("{}/user-agent".format(self.base_url))
    user_agent = response.json()['user-agent']
    return user_agent

  def get_headers(self):
    response = requests.get("{}/headers".format(self.base_url))
    headers = HeadersModel(response.json())
    return headers
```

该客户端有一个 base_url 类变量，用于存储待调用的基本 URL。get_ip()、get_user_agent()和 get_headers()方法的实现分别调用了 /ip、/user-agent 和/headers 端点，使用 base_url 变量构建了完整的 URL。

与 requests 库的交互和这三个方法非常相似。用 requests.get()调用端点，它将发送 HTTP

GET 请求至端点并返回一个响应对象。该对象有一些属性，如 status_code（存储响应代码）和 text（存储原始的响应数据），以及 json()方法。我们将用 json()方法将 JSON 响应导入一个字典。

IP 和用户代理都是简单的字符串，不过头文件对象稍微复杂一些，包含几个属性。因此，响应结果被包进一个自定义的 HeadersModel 对象中，定义如下所示。

```
class HeadersModel(object):

  def __init__(self, data):
    self.host = data['headers']['Host']
    self.user_agent = data['headers']['User-Agent']
    self.accept = data['headers']['Accept']
```

如果将前面的类存入名为 httpbin_client.py 的文件中，我们可以使用自定义的客户端，如下所示。

```
>>> from httpbin_client import HttpBinClient
>>> client = HttpBinClient()
>>> print(client.get_user_agent())
python-requests/2.8.1
>>> h = client.get_headers()
>>> print(h.host)
httpbin.org
```

为了 Web API 的简洁性，我们并没有在这个简单的任务中加入更复杂的应用层来误导你。实际上，可以用 requests 库从我们的应用直接与 Web API 交互。一方面，这种简单的交互对本例来说可以达到目的，但另一方面，社会媒体 API 在本质上非常具有动态性。新的端点会被加上，响应格式会改变，新的特征会不断被开发出来。例如，第 4 章的草稿刚刚完成时，Facebook 就引入了新的交互特征，这也导致了其 API 响应的变化。

换句话说，自定义的 Python 客户端加了一层抽象，允许我们只关注客户端的接口，而不是 Web API。从应用的角度来看，我们将关注客户端方法的调用，如 HttpBinClient.get_headers()，并隐藏/headers 端点在此类方法中发送响应的细节。如果来自 Web API 的响应格式未来发生改变，我们需要做的就是更新客户端并反映新的改变，但与 Web API 交互的所有应用不会受到影响，这多亏了这层额外的抽象。

httpbin.org 服务其实并不复杂，因此一个社会媒体 API 的实现不会很直接，但总的来说，核心要义是：用抽象层隐藏实现细节。这也是我们迄今使用的所有 Python 客户端的做法。



HTTP 状态码

为了更广泛地理解 HTTP 交互的工作方式，我们必须介绍响应状态码的重要性。来自 HTTP 服务器的请求在底层都是由数值状态码打开的，后面跟着描述该状态码的原因短语。例如，将浏览器指向特定的 URL 时，Web 服务器会用状态码 200 回应，并跟上原因短语 OK。然后该响应呈现我们请求的网页，并展示在浏览器中。Web 服务器还能以状态码 404 响应，并跟上原因短语 Not Found。浏览器会理解该请求并向用户返回合适的错误消息。

Web API 与前面介绍的简短交互没什么不同。根据不同的请求，服务器将发送一个成功响应（状态码 2xx）或一个错误（客户端错误码为 4xx，服务器错误码为 5xx）。客户端实现应该可以正确地翻译 Web API 提供的状态码，并进行相应的行动。还需要注意的是，有些服务采用的是非标准的状态码。例如，Twitter 的错误状态码是 420，Enhance your calm，在客户端发送过多请求、超过接入次数时使用。

状态码的完整列表及其含义参见 https://www.wikiwand.com/en/List_of_HTTP_status_codes，官方文档包含在 RFC 文档中。

有了开源库以后，很少需要从头实现一个客户端。到目前为止，你应该具备了着手实现客户端的能力。

8.6 小结

本章讨论了从社会媒体挖掘数据的更多选择。除了常用的流行社交网络，如 Twitter、Facebook 和 Google+，很多其他平台也提供了接入数据的 API。

当特定平台没有可用的 Python 客户端时，我们可以实现自定义的客户端，并利用 requests 库的简洁性。实现自定义客户端刚开始看起来可能增加了一些不必要的复杂性，但它提供了一个重要的抽象层，从长期来看是值得设计的。

除了自定义客户端的实现，我们还介绍了用于挖掘复杂对象的其他一些流行服务。我们检索的数据包括 YouTube 的视频、GitHub 的开源项目，以及 Yelp 的本地商家（如餐馆）。

本书未介绍的社会媒体平台还有很多，因为详尽的介绍是不现实的，但是我希望你能意识到还有很多可能性。

下一章是本书的最后一章。它介绍了语义网，并为你理解语义标注数据的重要性提供了概览。

第 9 章

关联数据和语义网



本章综述了语义网和相关技术。我们将介绍以下主题：

- ❑ 讨论作为数据网的语义网的基础
- ❑ 讨论微格式、链接数据和 RDF
- ❑ 从 DBpedia 挖掘语义关系
- ❑ 在 Google Maps 上绘制地理信息

9.1 数据网

万维网联盟（World Wide Web Consortium，W3C）是一个国际组织，它发布了 Web 标准并给出了语义网的定义。

> *"语义网就是数据网。"*

语义网及其定义都是由 Web 之父蒂姆 · 伯纳斯–李爵士创造的，他同时也是 W3C 的理事。当谈到自己的伟大发明时，他总是强调 Web 的社会影响，以及它为何更偏向一个社会创造而非技术创造（出自蒂姆 · 伯纳斯–李的《编织万维网》一书）。

如果简要分析 Web 的演进，Web 作为一个平台的前景就变得更加明确了。21 世纪头 10 年流行起来的一个关键词是 Web 2.0，它是在 20 世纪 90 年代末期创造出来的，后来由蒂姆 · 奥莱利传播开来。Web 2.0 意味着它是 Web 的一个新版本，但是它并不关注规格的技术性更新和变化，而是关注从旧的静态 Web 1.0 到以用户为中心理解 Web 并提供丰富用户体验的演进。

从 Web 1.0 到 Web 2.0 的演进可以用一个词概括——合作。蒂姆 · 伯纳斯–李并不认可这一点，因为 Web 2.0 用到的技术和标准与旧的 Web 1.0 时代相同。此外，用户的连接和协作始终是 Web 的主题。尽管有不同意见，但 Web 2.0 这个术语已经成为我们词汇的一部分，而且经常与社会网（social web，有时也用 Web 2.*X* 表示）这个名词同时出现。

那么应该将语义网放在哪个位置呢？根据 Web 创造者的观点，这应该是演进的下一步，即 Web 3.0。

Web 的自然演进是成为一个数据网。这个过程中至关重要的是数据模型和数据表示，它们要允许以机器理解的方式分享知识。通过使用以一致的语义格式分享的数据，机器可以支持用户的复杂信息需求和决策需求。

Web 中主流的文档格式是 HTML。这种格式是一种标记惯例，用于描述包含文本、多媒体对象（如图像或视频）以及文档间链接的文档的结构。

HTML 允许内容管理者在文档的头部指定一些元数据。这些信息不是用来展示的，因为浏览器通常只显示文档的正文。元数据标签包括作者的名字、版权信息、文档简介，以及描述文档的关键词。计算机可以翻译所有这些细节来分类文档。不过 HTML 的缺憾是不能指定更复杂的信息。

例如，通过使用 HTML 元数据，我们可以描述一个文档是关于 Woody Allen 的，但是该语言并不支持对复杂概念进行消歧，比如，该文档是关于 Woody Allen 在电影中饰演的角色、关于 Woody Allen 拍摄的电影，还是其他人导演的关于 Woody Allen 的电影？因为 HTML 的目的是描述文档的结构，所以它可能不是表示这种知识粒度的最佳方式。

另一方面，语义网技术允许我们进一步用实体、实体的属性以及实体间的关系来描述这个世界。举个例子，可以用表 9-1 来表示 Allen 的部分电影作品。

表 9-1　Woody Allen 的电影

艺术家	角色	电影
Woody Allen	导演	《曼哈顿》
Woody Allen	演员	《曼哈顿》
Woody Allen	导演	《赛末点》

表 9-1 显示了实体类型**艺术家**和**电影**间的链接关系。表示这种结构化知识属于语义网技术的范围。

接下来将综述有关语义网和知识表示的概念和术语。我们将介绍一些可以回答复杂查询的技术，如 Woody Allen 示例中的查询。

9.1.1　语义网词汇

本节主要介绍和回顾语义技术中常用的基本词汇。

标记语言（markup language）：这是用于标记文档的系统。给定一段文字或者说一段数据，标记语言允许内容管理者根据标记语言的特定含义标记文本或数据块。一个非常著名并广泛使用的标记语言就是 HTML。标记语言既包括展示性标记，也包括描述性标记。前者与文本显示的方式有关，后者提供了对标记数据的描述。在 HTML 的示例中，展示标签和描述标签都可以使用。

例如，<b>或<i>标签表示样式，内容设计者可以用它们将文本中特定的内容显示成粗体或斜体。另一方面，<strong>和<em>分别表示文本的特定部分应该被增强或强调。在普通的浏览器中，<b>和<strong>通常以粗体显示，而<i>和<em>以斜体显示。盲人用户无法在视觉上体会粗体样式的好处，但将短语标记成<strong>可以让屏幕阅读器理解应该如何阅读该短语。

语义标记（semantic markup）：正如前面介绍的，呈现事物的方式和理解事物的方式是不同的。语义标记描述的是所呈现信息的含义，而不是外观。在 HTML 和 XHTML 中，展示性标记标签并未被明确弃用，只是并不推荐使用。HTML5 与语义标记的概念更接近一些，在其中引入了一些语义标签，如<article>和<section>。同时，展示标签（如<b>和<i>）被保留且具有准确意义（也就是说，它们用于区别与普通文字样式的不同，不传递任何重要性）。

本体（ontology）：语义网的标志性技术之一就是本体。它表示特定领域的实体类型、属性和关系。本体的具体例子包括 WordNet 和 SNOMED CT，前者是一个词汇资源，其中的项根据含义分成了不同的概念组，而后者是一个医学词汇表。

维基百科显示了来自 WordNet 的一小段摘录。

```
dog, domestic dog, Canis familiaris
  => canine, canid
    => carnivore
      => placental, placental mammal, eutherian, eutherian mammal
        => mammal
          => vertebrate, craniate
            => chordate
              => animal, animate being, beast, brute, creature, fauna
                => ...
```

以上示例主要表示单词 dog 的含义。可以看到，为了表示一段知识，需要使用很多的层级来展示细节：狗并不简简单单是一种动物，还可以根据其在生物学上所属的特定科（犬科）或目（食肉目）进行分类。

可以用语义标记描述本体。例如，内容管理者可以使用**网络本体语言**（Web Ontology Language，OWL）来表示知识。构建本体的挑战包括要表示的领域的广阔性，以及人类知识和自然语言作为知识传播媒介所固有的歧义和不确定性。手动构建本体需要深入理解领域知识。自从哲学兴起以来，知识的描述和表示都一直是人类面临的基本问题之一，因此，我们可以将语义网本体看作哲学本体的实际应用。

分类法（taxonomy）：比本体的概念更窄一些，分类法指的是知识的层级表示。换句话说，它用于组织定义好的实体的类，类间的关系用 is a 和 has a 进行定义。本体和分类法的不同之处是，前者对更大范围的关系进行建模。

分众分类法（folksonomy）：也叫作社会分类法。建立分众分类法就是指进行社会化标注。用户可以用特定的标签来标记一段内容，而该标签是用户对一个特定类别的解读。该结果是分类

法的一种民主化，没有死板僵化的结构，能以用户的看法来展示信息。分众分类法的缺点是在现实中不可控且难以组织。它也可能表示一种特定的趋势，而非对特定领域的深入理解。

在社会媒体的环境中，分众分类法的一个典型案例就是 Twitter。用户用话题标签来标记其推文属于哪一种类型的话题。这样一来，其他用户可以搜索特定的主题或关注特定的事件。分众分类法的特征在 Twitter 话题标签使用上表现得非常显著。这些标签由推文的发布者选定，没有层级关系、没有组织，只有那些看起来有意义且被大众选择的标签才会成为趋势（民主化）。

推论和推理（inference and reasoning）：在逻辑领域中，推论是指从已知（或假设）的正确事实推导出逻辑结论的过程。另一个类似的术语是**推理**。理解这两个名词的一种方式是，将推论看作目标，推理看作实现方法。

推论的经典案例是苏格拉底三段论。我们思考一下以下事实（这些断言被认为是真的）：

- 凡人都是会死的
- 苏格拉底是人

根据以上前提，一个推理系统应该可以回答以下问题：**苏格拉底会死吗？**答案令人心惊：是。

三段论的通用模式如下所示：

- 每个 A 都是 B
- C 是 A
- 因此 C 是 B

这里的 A、B 和 C 可以是类别或个体。三段论是演绎推理的一个经典案例。通常来说，“由……推断”“推论自……”和“对……进行推理”都是该语境下的同义词，因为它们都需要从已知的事实中产生新的知识。

给定前面关于苏格拉底和其他人的知识，为了回答诸如**柏拉图会死吗**这样的问题，我们需要进一步学习以下内容。

闭合世界和开放世界假设（closed-world and open-world assumptions）：一个推理系统能够包括关于宇宙的闭合世界假设或者开放世界假设。这两种假设的关键差别在于，结论从（有限）已知事实集合中推论出来的方式。在基于闭合世界假设的系统中，每个未知的事实都是假的；而在基于开放世界假设的系统中，未知的事实可能为真也可能为假，因此不强制把未知解读为假。这样它们可以处理不完整的知识，也就是部分指定或者会随时间完善的知识库。

例如，典型的逻辑编程语言 Prolog（来源于 programming in logic）是基于闭合世界假设的。有关苏格拉底的知识库用 Prolog 的语法表示如下所示：

```
mortal(X) :- man(X).
man(socrates).
```

关于柏拉图是否会死的答案是：否。这是因为系统没有关于柏拉图的任何知识，因此不能做进一步的推论。

闭合世界假设和开放世界假设的二重性也影响了不同知识库的融合方式。如果两个知识库提供了相反的事实，基于闭合世界的系统将触发错误，因为该系统不能处理这种类型的不一致。另一方面，基于开放世界假设的系统将产生更多的知识并尝试找到解决不一致的方法，从而尝试从不一致中恢复。一种方法是为事实分配概率：简化起见，两个相互矛盾的事实可以各有 50%的概率为真。

闭合世界假设的一个示例是机票预订数据库。该系统对于特定的领域有完整的信息。这样回答某些问题就非常清楚了：如果座位是在值机时分配，而一名已经预订该航班的乘客并没有分配座位，那么就能推断出该乘客还没有值机，从而可以向该乘客发送一封电子邮件以提醒他在线值机。

另一方面，当一个系统不具备特定领域的完整信息时，可以应用开放世界假设。例如，基于 Web 的在线职位发布平台知道给定工作的具体位置。如果一家公司发布位于纽约的职位，但收到了欧洲求职者的应聘申请，那么该系统无法基于现有的信息推断该应试者能否到美国工作。因此，此类系统可能需要在应聘者提出申请时明确询问。

逻辑和逻辑编程是非常广阔的研究领域，因此，在本节详述该主题的方方面面是不现实的。本章的目的仅仅是了解该领域。对于 Web 来说，假设知识库不完整通常是更保险的做法，因为 Web 是一个具有不完整信息的系统。实际上，RDF（resource description framework，资源描述框架）等框架是基于开放世界假设的。

9.1.2 微格式

微格式扩展了 HTML，允许作者指定机器可读的语义标注，这些标注可以是人物、机构、事件和很多不同类型的对象。它们是基本的惯例，为内容生产者提供将无歧义的结构化数据嵌入网页的机会。最新的进展融合进了 microformat2。

有趣的微格式示例如下所示。

- `h-card`：表示人物、机构和相关的联络信息
- `h-event`：表示事件信息，如地点和开始时间
- `h-geo`：表示地理坐标，与 `h-card` 和 `h-event` 结合使用
- XHTML Friends Network（XFN）：通过超链接表示人际关系
- `h-resume`：用于在 Web 上发布简历信息和简历，包括教育、经历和技能
- `h-product`：描述与产品相关的信息，如品牌、价格或描述
- `h-recipe`：表示 Web 上的食谱，包括原料、数量和指导
- `h-review`：发布对任何项（如 `h-card`、`h-event`、`h-geo` 或者 `h-product`）的评论，

包括评分信息和描述

以下代码片段包括一个 XFN 标记的示例，该标记用于通过超链接表示人际关系：

```
<div>
  <a href="http://marcobonzanini.com" rel="me">Marco<a>
  <a href="http://example.org/Peter" rel="friend met">Peter<a>
  <a href="http://example.org/Mary" rel="friend met">Mary<a>
  <a href="http://example.org/John" rel="friend">John<a>
</div>
```

XFN 通过 `rel` 属性增强了普通的 HTML。一些 HTML 链接已经根据人际关系被打上了标签。第一个链接标签为 `me`，意思是该链接与文档的作者相关。另一个链接被标记为 `friend` 和 `met` 类，后者表示见过的人（与只在线联系的关系相反）。其他一些类还包括 `spouse`、`child`、`parent`、`acquaintance` 和 `co-worker`。

以下代码片段用 `h-geo` 标记表示伦敦的地理坐标：

```
<p class="h-geo">
  <span class="p-latitude">51.50722</span>,
  <span class="p-longitude">-0.12750</span>
</p>
```

`h-geo` 是微格式的最新版本，但旧版本（`geo`）仍然在被广泛使用。可以用以下旧格式重写以上代码：

```
<div class="geo">
  <span class="latitude">51.50722</span>,
  <span class="longitude">-0.12750</span>
</div>
```

如果以上微格式的示例对你来说非常简单易懂，那是因为微格式本意如此。微格式的微表示了其特点：极其关注非常具体的领域。每个微格式规范只在限定的领域解决一个定义好的问题（例如，描述人际关系或者提供地理坐标）。这一特点使得微格式具有高度的可组合性，一个复杂的文档可以用多个微格式来提供丰富的信息。此外，一些微格式可以嵌入其他微格式来提供更丰富的数据。例如，`h-card` 可以包括 `h-geo`，`h-event` 可以包括 `h-card` 和 `h-geo`，等等。

9.1.3　关联数据和开放数据

关联数据是指用 W3C 标准在 Web 上发布结构化数据的一组原则，有助于专门的算法来探索数据间的关系。

语义网并不仅仅是将数据放在 Web 上，它还允许人们（和机器）利用和探索数据。伯纳斯-李因此提出了一组原则来推动关联数据的公开，并推动其向“关联开放数据”发展，也就是可以通过开放许可来获取的关联数据。

Web 实际上是通过超链接连接的文档网络，与之类似，数据网同样基于发布在 Web 上的文档及其链接。不过，数据网是关于最通用形式的数据，而不是关于文档。用来描述任意**事物**的格式是 RDF，而不是(X)HTML。

以下是伯纳斯–李提出的四个原则，可以用作构建关联数据的指南：

- 将 URI 作为事物的名称
- 使用 HTTP URI，这样人们可以查找这些名字
- 当人们查找一个 URI 时，用标准（RDF*、SPARQL）提供有用的信息
- 保留到其他 URI 的链接，这样人们可以发现更多事物

这些基本原则提供了对内容发布者的期望，以便能产生关联数据。

为了鼓励人们接受关联数据，一个五星评分系统被开发出来，便于人们评估自己的关联数据并理解关联开放数据的重要性。这特别针对与政府相关的数据拥有者，可以推动数据的透明度。对五星评分系统的描述如下所示。

- **一星**：表示在 Web 上可用（任何格式），但是要有开放许可才能成为开放数据。
- **二星**：表示作为机器可读的结构化数据（如 Excel，而不是表格的图像扫描）可用。
- **三星**：满足二星的条件，而且是非专用格式（如 CSV，而不是 Excel）。
- **四星**：满足前面所有星级的条件，而且使用来自 W3C 的开放标准（RDF 和 SPARQL）来识别事物，从而让人们可以识别你的东西。
- **五星**：满足前面所有星级的条件，而且将你的数据与其他人的数据进行链接以提供上下文。

达到五星后，数据集的下一步是提供额外的元数据（关于数据的数据）。对于政府数据，要列在主要数据目录上：美国的是 https://www.data.gov/，英国的是 https://data.gov.uk/，欧盟的是 http://data.europa.eu/euodp/en/data。

显而易见，关联数据是语义网的基本概念。实现关联数据的一个著名数据集示例是 DBpedia，该项目的目标是从维基百科发布的信息中抽取结构化知识。

9.1.4　RDF

RDF 来源于 W3C 规范家族，最初的设计目的是作为元数据模型，用于对结构化知识进行建模。

与在数据库设计领域非常流行的经典概念性模型框架（如实体关系模型）类似，RDF 也是基于对资源的声明。这样的声明是用**三元组**表示的，因为它有三个基本组成成分：主语、谓语和宾语。

以下是主谓宾三元组表示的一个示例：

```
(Rome, capital_of, Italy)
(Madrid, capital_of, Spain)
(Peter, friend_of, Mary)
(Mary, friend_of, Peter)
(Mary, lives_in, Rome)
(Peter, lives_in, Madrid)
```

以上语法是任意的，并不遵循特定的 RDF 规范，但它可以作为理解数据模型简洁性的示例。基于三元组存储声明的方法可以表示任意知识。可以自然地把一个 RDF 声明的集合看作一个图，并且是有标签的有向多重图：有标签是因为节点（资源）和边（谓语）有相应的信息，如边的名称；有向是因为主谓宾关系有明确的方向；多重图是因为同样的节点间可能有多个并行的关系，也就是说，两个相同的资源可能处于不同的关系中，而这些关系都带有不同的语义。

图 9-1 是前面示例的可视化表示。

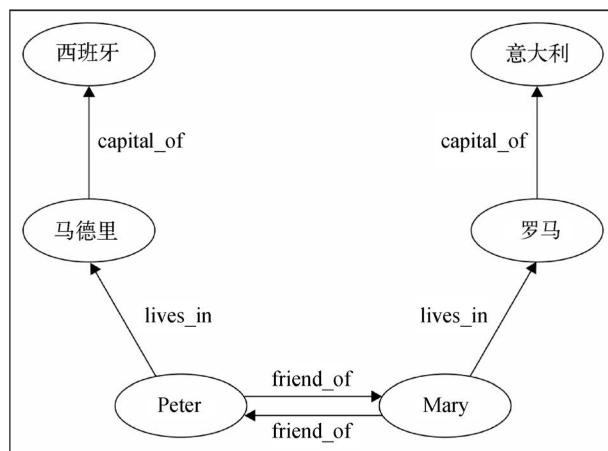


图 9-1　三元组知识的可视化表示

下一层的知识表示在 RDF 中，它能让我们定义三元组的一些额外结构。谓语 `rdf:type` 允许我们声明特定的对象属于特定的类型。此外，RDF Schema（RDFS）允许我们定义不同类之间的关系。OWL 是一个相关的技术，允许我们表达不同类之间的关系，但它所提供关于数据模型的信息在约束和标注方面更加丰富。

RDF 可以序列化（即导出）为多种不同的格式。XML 可能是最流行的格式，还有其他格式，例如 N3。 这也可以说明 RDF 是一种用三元组表示知识的方式，而不只是一种文件格式。

9.1.5　JSON-LD 格式

我们在前面的几章中都碰到了 JSON 格式，并看到了其灵活性。JSON 和关联数据的关系是用 JSON-LD 格式表示的。这是一种基于 JSON 的轻量级数据格式，因此更易于人和机器读写。

当从不同的数据源链接数据时，我们可能会面临一个歧义问题。思考一下表示同一个人的两个 JSON 文档，如下所示：

```
/* 文档 1 */
{
  "name": "Marco",
  "homepage": "http://marcobonzanini.com"
}
/* 文档 2 */
{
  "name": "mb123",
  "homepage": "http://marcobonzanini.com"
}
```

可以看到，以上示例中的 Schema 是一样的：两个文档都有 `name` 和 `homepage` 属性。歧义来自两个数据源使用相同的属性名称，但是具体含义不同。实际上，第一个文档用 `name` 指人名，而第二个文档用 `name` 表示登录名。

我们感觉这两个文档指的是同一个实体，因为 `homepage` 属性的值相同，但在没有进一步信息的情况下，无法解决这个歧义问题。

JSON-LD 引入了一个名为**上下文**的简单概念。当使用上下文时，我们可以通过已经熟悉的 URL 解决以上歧义问题。

查看来自 JSON-LD 网站的以下代码片段：

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

上述文档将`@context` 属性作为对自身有效属性及其含义列表的引用。这样一来，就能确定我们期望在文档中看到的是哪种属性了。

以上示例还提到了 JSON-LD 的另一个特殊属性`@id`。该属性是一个全局标识符。不同的应用在表示相同实体时可以使用该全局标识符。这样一来，来自不同数据源的数据可以消除歧义并链接起来。例如，一个应用可以用标准 Schema 定义一个人，并用附加属性进行增强。

上下文（属性的列表和含义）和实体的全局识别方式是关联数据的基础。JSON-LD 格式用一种简单且优雅的方式解决了这个问题。

9.1.6　Schema.org

发布于 2011 年，Schema.org 是一些主流搜索引擎（Google、Bing 和 Yahoo!，Yandex 随后加

入）的合作成果。他们合作的初衷是创建一个共享的 Schema 集合，描述用于网页的结构化标记数据。

该提议关于使用 Schema.org 词汇及微格式、RDF 格式或 JSON-LD 来用网站的元数据增强网站的内容。通过使用提议的 Schema，内容管理者可以用额外的知识标记他们的网站，从而帮助搜索引擎和其他解析器更好地理解网站内容。

关于机构和人物的 Schema 已经被用于影响 Google 知识图谱这样的系统。Google 知识图谱是一个知识库，可以在搜索特定实体时用语义信息增强 Google 搜索引擎的结果。

9.2　从 DBpedia 挖掘关系

DBpedia 是最受欢迎的关联数据源之一。它基于维基百科，用实体间的语义关系增强这一流行的维基式百科全书的内容。可以用类 SQL 的语言 SPARQL 从网站中获取 DBpedia 的结构化信息，SPARQL 是一种针对 RDF 的语义查询语言。

在 Python 中，我们可以用 SPARQL 查询数据库，也可以使用 RDFLib 包，它是一个用于 RDF 的库。

可以用 `pip` 将 RDFLib 安装在虚拟环境中。

```
$ pip install rdflib
```

鉴于语义网主题非常复杂，我们用一个示例来展示 DBpedia 的能力，同时了解如何使用 RDFLib 包。

`rdf_summarize_entity.py` 脚本查询给定实体并尝试将其摘要输出给用户。

```
# Chap09/rdf_summarize_entity.py
from argparse import ArgumentParser
import rdflib

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--entity')
  return parser

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()
  entity_url = 'http://dbpedia.org/resource/{}'.format(args.entity)

  g = rdflib.Graph()
  g.parse(entity_url)

  disambiguate_url = 'http://dbpedia.org/ontology/wikiPageDisambiguates'
```

```
    query = (rdflib.URIRef(entity_url),
             rdflib.URIRef(disambiguate_url),
             None)
    disambiguate = list(g.triples(query))
    if len(disambiguate) > 1:
      print("The resource {}:".format(entity_url))
      for subj, pred, obj in disambiguate:
        print('... may refer to: {}'.format(obj))
    else:
      query = (rdflib.URIRef(entity_url),
               rdflib.URIRef('http://dbpedia.org/ontology/abstract'),
               None)
      abstract = list(g.triples(query))
      for subj, pred, obj in abstract:
        if obj.language == 'en':
          print(obj)
```

脚本中 ArgumentParser 的功能与之前示例中相同。--entity 参数用于将实体名称传递给脚本。该脚本根据 dbpedia.org/resource/<entity-name>模式构建了 DBpedia 上的实体 URL。该 URL 用 rdflib.Graph 类构建了一个 RDF 图。

下一步确定是否需要消除歧义，即查看是否有给定名称指向多个实体。这是通过查询图中的三元组实现的：由给定实体的 URL 作为主语，wikiPageDisambiguates 关系作为谓语，且宾语为空。为了构建主语、谓语和宾语，需要使用 rdflib.URIRef 类，它以实体或关系的 URL 作为唯一的参数。None 对象是三元组的一个元素，triples()方法将其作为一个占位符，用于任何可能的值（它实质上是一个通配符）。

找到歧义信息后，该列表会被打印出来给用户。否则，该实体会被查找，特别是会用 triples()方法检索其摘要，然后给出 None 对象。给定的实体可能有多个摘要，因为实体的内容也许是多语言的。迭代结果时，我们可以只打印英文版本（标签为 en）。

例如，可以用以下脚本对 Python 语言做摘要描述。

```
$ python rdf_summarize_entity.py \
  --entity "Python"
```

只将 Python 作为关键词搜索维基百科不会直接得到我们期望的结果，因为这个词本身是有歧义的。实际上，好几个实体都可能匹配这个词，因此代码会打印出完整列表。以下是输出的一个小片段。

```
The resource http://dbpedia.org/resource/Python:
... may refer to: http://dbpedia.org/resource/Python_(film)
... may refer to: http://dbpedia.org/resource/
Python_(Coney_Island,_Cincinnati,_Ohio)
... may refer to: http://dbpedia.org/resource/Python_(genus)
... may refer to: http://dbpedia.org/resource/Python_(programming_language)
# (more disambiguations)
```

准确识别 Python_(programming_language)实体后，可以用准确的实体名称重新运行脚本。

```
$ python rdf_summarize_entity.py \
  --entity "Python_(programming_language)"
```

这次的输出就是我们所期望的。一个简短的输出片段如下所示。

```
Python is a widely used general-purpose, high-level programming language.
Its design philosophy emphasizes code readability, (snip)
```

该示例表明，将基于三元组的高层次 RDF 概念模型与 Python 接口结合起来是一个相对直接的任务。

9.3 挖掘地理坐标

前面介绍过，geo 和 h-geo 是用于发布地理信息的微格式。当阅读关于社会媒体的图书时，有人可能会问地理元数据是否为对社交数据的描述。当分析地理数据时，关键是让不同的应用都可以探索地理信息。例如，用户可能希望做基于位置的商家搜索（如在 Yelp 应用中）或者查找在特定地点拍摄的照片。更宽泛地说，应用场景还有每个人的物理位置或者寻找处于某个位置的物体。地理元数据允许应用实现与地理相关的定制。在社交和移动数据的时代，这些定制为应用开发者打开了新的思路。

回到对语义标记数据的介绍，本节将描述如何用维基百科从网页中抽取地理元数据。作为提醒，经典的 geo 微框架如下所示。

```
<p class="geo">
  <span class="latitude">51.50722</span>,
  <span class="longitude">-0.12750</span>
</p>
```

新版本中的 h-geo 也是类似的，只是类的名称有所改变。一些维基百科模板还会用到不同的格式，如下所示。

```
<span class="geo">51.50722; -0.12750</span>
```

为了从维基百科抽取该信息，需要执行两个步骤：首先检索页面并解析 HTML 代码，然后寻找类的名称。下一节将对此过程进行详细介绍。

9.3.1 从维基百科抽取地理数据

从头开始实现从网页抽取地理数据的过程非常简单，比如用我们已经介绍过的 requests 和 Beautiful Soup 库。尽管有这样的简便方法，但还是有专门的 Python 包 mf2py 来解决这个特定的问题。它提供了一个简单的接口，可以快速解析给定 URL 的网页资源。这个库还提供了一些

辅助函数，可以将解析数据从 Python 字典移到 JSON 字符串中。

可以用 pip 将 mf2py 安装到虚拟环境中。

```
$ pip install mf2py
```

这个库提供了三种方法来解析语义标记数据：我们可以将一段内容作为字符串、文件指针或 URL 传递给 parse()函数。以下示例展示了如何解析带有语义标记的字符串。

```
>>> import mf2py
>>> content = '<span class="geo">51.50722; -0.12750</span>'
>>> obj = mf2py.parse(doc=content)
```

现在 obj 变量包含一个解析内容的字典。我们可以将其转换成 JSON 字符串并漂亮地打印出来。

```
>>> import json
>>> print(json.dumps(obj, indent=2))
{
  "items": [
  {
    "type": [
      "h-geo"
    ],
    "properties": {
      "name": [
        "51.50722; -0.12750"
      ]
    }
  }
  ],
  "rels": {},
  "rel-urls": {}
}
```

如果希望将一个文件指针传递给解析函数，过程非常相似，如下所示。

```
>>> with open('some_content.xml') as fp:
...   obj = mf2py.parse(doc=fp)
```

如果希望将一个 URL 传递给解析函数，那么需要用 url 参数，而不是 doc 参数，如下所示。

```
>>> obj = mf2py.parse(url='http://example.com/your-url')
```

mf2py 库的解析功能是基于 Beautiful Soup 实现的。如果调用 Beautiful Soup 时没有特别指定解析器，那么 mf2py 将尝试最佳选项。例如，如果我们安装了 lxml（第 6 章中已经实践过），那么它就会是一个最佳选择。调用 parse()函数会触发以下警告：

```
UserWarning: No parser was explicitly specified, so I'm using the best
available HTML parser for this system ("lxml"). This usually isn't a
problem, but if you run this code on another system, or in a different
```

```
virtual environment, it may use a different parser and behave differently.
To get rid of this warning, change this:

 BeautifulSoup([your markup])

to this:

 BeautifulSoup([your markup], "lxml")
```

以上消息由 Beautiful Soup 抛出，但我们并不是直接与这个库交互，因此如何解决这个问题令人困惑。解决方式是在调用 `parse()` 函数时显式指定解析器的名称，如下所示。

```
>>> obj = mf2py.parse(doc=content, html_parser='lxml')
```

介绍了 mf2py 后，我们可以融会贯通，尝试解析维基百科页面中的地理信息。

`micro_geo_wiki.py` 脚本以维基百科 URL 作为输入并显示相应坐标的地点。

```
# Chap09/micro_geo_wiki.py
from argparse import ArgumentParser
import mf2py

def get_parser():
  parser = ArgumentParser()
  parser.add_argument('--url')
  return parser

def get_geo(doc):
  coords = []
  for d in doc['items']:
    try:
      data = {
        'name': d['properties']['name'][0],
        'geo': d['properties']['geo'][0]['value']
      }
      coords.append(data)
    except (IndexError, KeyError):
      pass
  return coords

if __name__ == '__main__':
  parser = get_parser()
  args = parser.parse_args()

  doc = mf2py.parse(url=args.url, html_parser='lxml')
  coords = get_geo(doc)
  for item in coords:
    print(item)
```

上述脚本用 `ArgumentParser` 从命令行获取输入参数。`--url` 参数用于传递我们希望解析的 URL。

接下来调用 `mf2py.parse()` 函数，传递 URL 作为参数，获得微格式信息的字典。

抽取地理坐标的核心逻辑由 `get_geo()` 函数实现，它将解析字典作为输入并返回字典列表作为输出。输出中的每个字典有两个键：`name`（地点名）和 `geo`（坐标）。

可以用以下命令运行以上脚本。

```
$ python micro_geo_wiki.py \
  --url "https://en.wikipedia.org/wiki/London"
```

输出结果如下所示。

```
{'name': 'London', 'geo': '51.50722; -0.12750'}
```

不出意料的是，在关于伦敦的维基百科页面，我们找到的唯一坐标是伦敦市中心的坐标。

维基百科还提供了许多包含很多地点的页面，如下所示。

```
$ python micro_geo_wiki.py --url \
"https://en.wikipedia.org/wiki/List_of_United_States_cities_by_population"
```

运行上述命令会生成很长的输出（超过 300 行），输出结果包含一些美国城市的地理信息。以下是输出的一个小片段。

```
{'geo': '40.6643; -73.9385', 'name': '1 New York City'}
{'geo': '34.0194; -118.4108', 'name': '2 Los Angeles'}
{'geo': '41.8376; -87.6818', 'name': '3 Chicago'}
# (snip)
```

该示例表明，从网页抽取地理数据并不是特别困难的事情。mf2py 库允许我们通过短短几行代码来完成任务。

下一节会更深入一步：获得地理信息后，我们可以做什么呢？将其绘制在地图上可能是最直接的答案，因此我们将用 Google Maps 将地理数据可视化。

9.3.2 在 Google Maps 上绘制地理数据

Google Maps 是一项非常受欢迎的服务，并不需要过多介绍。它功能众多，其中就有用感兴趣的点创建自定义地图的功能。

要自动创建地图并确保不同应用间的互操作性，一种方法是用常用格式分享坐标。本节将介绍 Keyhole 标记语言（keyhole markup language，KML），它可以将数据导成 Google Maps 可识别的格式。KML 格式是一种 XML 标记，用于表达二维和三维的地图地理坐标数据。它最初是为 Google Earth 所开发的，现应用于很多领域。

以下片段是 KML 文档的示例，它表示地图中伦敦的位置。

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <Placemark>
      <name>London</name>
      <description>London</description>
      <Point>
        <coordinates>-0.12750,51.50722</coordinates>
      </Point>
    </Placemark>
  </Document>
</kml>
```

我们可以观察 `coordinates` 标签的值。从维基百科中抽取出的地理数据格式为 `latitude;longitude`，而 KML 标记使用的是 `longitude, latitude, altitude`，这里海拔参数是可选的，没有给出时默认为 0。

为了在 Google Maps 上对地理坐标列表进行可视化，接下来的任务是生成 KML 格式的列表。在 Python 中，PyKML 库可以简化该过程。

可以用 `pip` 将这个库安装到虚拟环境中。

```
$ pip install pykml
```

以下的 `micro_geo2kml.py` 脚本扩展了前面从维基百科抽取地理数据的例子，将结果输出为 KML 格式的文件。



这个脚本尝试用 lxml 库将 XML 序列化为一个字符串。如果这个库不存在，它将使用一个 Python 标准库——ElementTree。有关 lxml 库的详细介绍，可以参见第 6 章。

```
# Chap09/micro_geo2kml.py
from argparse import ArgumentParser
import mf2py
from pykml.factory import KML_ElementMaker as KML

  try:
    from lxml import etree
  except ImportError:
    import xml.etree.ElementTree as etree

  def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--url')
    parser.add_argument('--output')
    parser.add_argument('--n', default=20)
    return parser

  def get_geo(doc):
    coords = []
    for d in doc['items']:
    try:
```

```
      data = {
        'name': d['properties']['name'][0],
          'geo': d['properties']['geo'][0]['value']
        }
        coords.append(data)
      except (IndexError, KeyError):
        pass
    return coords

  if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    doc = mf2py.parse(url=args.url)
    coords = get_geo(doc)
    folder = KML.Folder()
    for item in coords[:args.n]:
      lat, lon = item['geo'].split('; ')
      place_coords = ','.join([lon, lat])
      place = KML.Placemark(
                          KML.name(item['name']),
                          KML.Point(KML.coordinates(place_coords))
                          )
      folder.append(place)

    with open(args.output, 'w') as fout:
      xml = etree.tostring(folder,
                         pretty_print=True).decode('utf8')
      fout.write(xml)
```

上述脚本用 ArgumentParser 从命令行获取三个输入参数：--url 参数用于获取待解析的网页；--output 用于指定保存地理信息的 KML 文件的名称；可选参数--n 默认为 20，用于指定我们希望在地图中包含多少个地点。

在主模块中，我们将解析给定的页面并用 get_geo()函数获取地理坐标。函数的返回值是一个字典列表，每个字典包含作为键的 name 和 geo。在 KML 术语中，地点称为**地点标记**，地点列表称为**文件夹**。换句话说，该脚本将 Python 字典列表翻译为一份装有地点标记的文件夹。

这个任务的执行是通过将 folder 变量初始化为一个空的 KML.Folder 对象。当用地理信息迭代字典列表时，我们将为列表中的每个元素创建一个 KML.Placemark 对象。地点标记由一个 KML.name 和 KML.Point 组成，即拥有地理坐标信息的对象。

为了用 KML 格式创建坐标，我们需要分隔坐标字符串。坐标的原始格式是 latitude; longitude，分隔后交换这两个值并用逗号代替分号，最终的字符串存储在 place_coords 变量中，格式为 longitude, latitude。

创建好地点标记列表后，最后一步是将这个对象保存为 XML 格式。这一步是由 toString()方法实现的，它返回一个 bytes 对象，因此需要在写入文件前解码。

可以用以下命令执行上述脚本。

```
$ python micro_geo2kml.py \
  --url "https://en.wikipedia.org /wiki/
    List_of_United_States_cities_by_population" \
  --output us_cities.kml \
  --n 20
```

现在地理数据保存在 us_cities.kml 文件中，Google Maps 可以导入该文件来创建一个自定义地图。

在 Google Maps 的菜单中选择 Your places 来查看我们喜欢的地点，然后选择 MAPS 打开自定义地图列表。图 9-2 展示了菜单及自定义地图的列表。

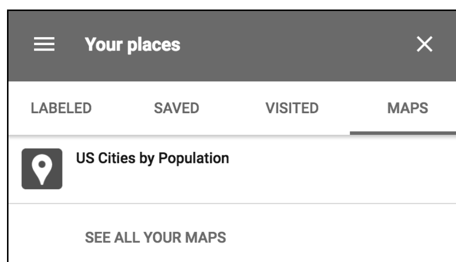


图 9-2　Google Maps 中的自定义地图列表

点击 CREATE MAP 后，我们可以从文件中导入数据，导入的文件格式可以是表格、CSV 或 KML 文件。选择脚本生成的 us_cities.kml 文件后，可以看到如图 9-3 所示的结果。

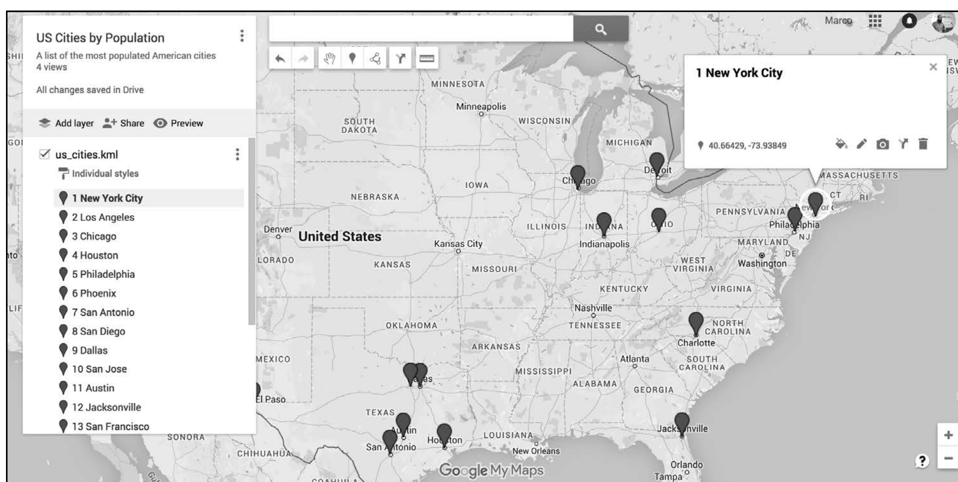


图 9-3　Google Maps 显示的美国城市列表（按人口排序）

每个地点标记由地图中的红色大头针标记，点击大头针会突出显示详细信息（本例中为名称和坐标）。

9.4　小结

本章开头介绍了 Web 之父蒂姆·伯纳斯–李的愿景。将 Web 当作社会产物而非技术产物的观点可以帮助我们模糊社交数据和语义数据的边界。

我们还介绍了语义网的宏大蓝图，以及这些年提出的不同技术如何使其变得更加现实和牢固。虽然在过去的 15~20 年中，人们对语义网大肆宣传、期望颇高，但是蒂姆·伯纳斯–李的愿景尚未普及。

以 W3C、政府和其他私人企业（如 Schema.org）为代表的社区努力推动了这一方向的发展。尽管质疑者有很多材料来批评语义网的诺言尚未完全实现，但是我们更愿意关注目前的机会。

社会媒体生态系统的当前趋势表明，社交数据和语义数据有很深的关联。本书介绍了从不同的社会媒体平台挖掘数据的方法，主要是用平台提供的 API 以及一些用于数据挖掘和数据分析的 Python 流行工具。单凭本书只能对该主题的皮毛做一些介绍，我们希望你可以意识到，在社会媒体的背景下，数据挖掘实践者和学者有非常多的机会。

版权声明



微信连接



回复“大数据”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

Python是非常适合数据科学家在中小型数据集上建造原型、可视化和分析数据的编程语言。本书将帮助你获取和分析来自各大社会媒体网站的数据，展示如何使用科学的Python工具来挖掘Stack Exchange等流行社交网站。

- 通过公共API与社会媒体平台进行交互
- 以方便的格式存储社交数据以进行分析
- 使用Python数据科学工具对社交数据进行切片和分段
- 应用文本分析技术来理解人们在社会媒体上谈论的内容
- 运用先进的统计和分析技术从数据中发现有用的见解
- 采用Web技术进行可视化，以探索数据和呈现数据产品

[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/数据挖掘

人民邮电出版社网址：www.ptpress.com.cn


ISBN 978-7-115-49401-6

定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks